Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

Tiago Fernando Alves de Freitas

Práticas Ágeis no Desenvolvimento de Sistemas Embutidos

Agile Practices in Embedded Systems Development

Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

Tiago Fernando Alves de Freitas

Práticas Ágeis no Desenvolvimento de Sistemas Embutidos

Agile Practices in Embedded Systems Development

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Pedro Nicolau Faria da Fonseca, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Ilídio Fernando de Castro Oliveira, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president	Prof. Doutor Paulo Bacelar Reis Pedreiras Professor Auxiliar da Universidade de Aveiro
vogais / examiners committee	Prof. Doutor Luís Miguel Pinho de Almeida Professor Associado da Faculdade de Engenharia da Universidade do Porto
	Prof. Doutor Pedro Nicolau Faria da Fonseca Professor Auxiliar da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Agradeço aos meus orientadores Pedro Fonseca e Ilídio Oliveira pelo tempo, disponibilidade e paciência que tiveram comigo. Quero agradecer também à empresa Exatronic pela disponibilização de espaço e meios que tornaram possível o desenvolvimento desta dissertação.

Agradeço também a todos os que se cruzaram comigo no teatro, ao GrETUA, à minha equipa de voleibol e todos os meus colegas de curso por terem me ajudado e acreditado em mim.

Por fim, quero agradecer à minha família pelo enorme apoio e confiança e a todos os meus amigos mais próximos: Anderson, Cide, David C., David S., Mascarenhas, Max, Paulinho, Vânia e Victor.

palavras-chave

Sistemas embutidos, metodologias ágeis, integração contínua, desenvolvimento orientado por testes, ferramentas de análise estática, processos de desenvolvimento, Atmel, CppUTest, Jenkins, Cygwin, Testes Unitários.

resumo

As metodologias ágeis ganharam popularidade depois de um grupo de profissionais em diferentes métodos de desenvolvimento se juntar e criar um manifesto ágil. Estas metodologias foram criadas com o intuito de melhorar a forma de desenvolver software, tendo como foco principal a satisfação do cliente. Cada vez mais estão a ser usadas em diversos projetos substituindo a abordagem mais tradicional que atualmente ainda está muito presente. Por exemplo, *Waterfall* é uma metodologia tradicional onde todo o desenvolvimento é planeado deixando pouco espaço para alterações por parte do cliente. O interesse das empresas nestas metodologias tem aumentado. As empresas querem saber mais sobre esta nova forma de desenvolvimento e quais as vantagens que estas vão ter comparando com o seu atual método de desenvolvimento, que geralmente é um método tradicional. A aplicação de metodologias ágeis na área de programação para sistemas embutidos é diferente dos sistemas de informação. O desenvolvimento deste tipo de sistemas tem de ter em conta a parte do hardware e software.

No contexto da empresa Exatronic, esta dissertação tem como objetivo investigar a abordagem ágil de forma a recomendar práticas que podem ser adaptadas por esta empresa e com elas obter melhores resultados. A empresa disponibilizou um projeto já terminado para as praticas escolhidas serem aplicadas e simuladas, tendo em conta a plataforma de desenvolvimento *Atmel Studio* e tipo de processadores *Atmel* usados pela empresa. As práticas recomendadas foram duas, integração contínua e desenvolvimento orientado por testes, pois são as únicas onde é possível criar um ambiente para a sua utilização e simulação. Por fim, são analisadas as vantagens do uso destas praticas no projeto da empresa.

keywords

Embedded systems, agile methodologies, continuous integration, test driven development, static analysis tools, development process, Atmel, CppUTest, Jenkins, Cygwin, Unit Tests.

abstract

Agile methodologies gained popularity after a group of professionals in different development methods join and create an agile manifesto. These methodologies were created in order to improve the way to develop software, focusing mainly on customer satisfaction. They are increasingly being used in several projects, replacing the more traditional approach currently very present. For example, Waterfall is a traditional approach in which all development is planned, leaving little space for customer changes. The interest of companies in these methodologies has increased. Companies want to know more about this new way of development and what advantages they will have compared to their current development method, which is usually a traditional one. The application of agile methodologies in embedded systems is different from the informational systems. The development of such systems has to take into account the part of hardware and software.

In the context of Exatronic company, this dissertation aims to investigate the agile approach in order to recommend practices that could be adapted in the company. Exatronic provided a finished project for the selected practices be implemented and simulated, taking into account the Atmel Studio development platform and the Atmel processors used by the company. The recommended practices were two, continuous integration and test driven development, because they are the only ones where is possible to create an environment for its use and simulation.

Contents

List of Figures

List of Tables

List of Acronyms

CI
Continuous Integration
TDD
Test Driven Development
MoSCoW
Musts, Shoulds, Coulds and Won't haves
FDD
Feature-driven development
XP
Extreme Programing
ATM
Automated Teller Machine
MP3
MPEG-1/2 Audio Layer 3
SW
Software
HW
Hardware
HDL
Hardware Description Language
VHDL
VHSIC Hardware Description Language
UML
Unified Modeling Language
SVN
Subversion
ARM
Advanced RISC Machine
GNU
GNU's Not Unix! GNU
LTA
Local Test Automation
SCM
Source Code Management
CPD
Copy/Paste Detector
OS
Operating System
XML
Extensible Markup Language
GCC
GNU Compiler Collection
ASF
Atmel Software Framework
SFR
Special Function Registers
CPU
Central Processing Unit
POSIX
Portable Operating System Interface
I/O
Input / Output
IDE
Integrated Development Environment
LED
Light Emitting Diode

chapter 1

Introduction

1.1 Motivation

Nowadays all companies are competing to be more successful in the market. However, to achieve this, they need to deliver quality products and meet deadlines. Agile methodologies try to change developers' minds by using non-traditional practices like *Waterfall* to deliver better products in the shortest time. These methodologies promise to reduce development time, improve communication within the team and between team and costumer, and deliver a cleaner code with better quality in time. Yet, these practices are seen as hard to implement, especially for embedded development which has some dependencies and limitations compared to informational systems. It is necessary to take into consideration not only the software but also the hardware part and there is also the performance constraints to respect. Regarding the development of embedded systems, there is a big interest in trying to adapt agile methodologies, or at least some practices. If we look around, we are surrounded by microprocessors, and there is a lot of competition in this business. There is a need for the embedded world to improve their products in order to survive in the market.

Exatronic [1], is a company specialized in research, design, development and industrialization of innovative solutions in Information Technology, Communication and Electronics. It has been In the market since 1995 and its mission is to deliver innovative solutions with integrated electronics for the business and/or products for customers, adding value to their ideas. It is my aim in this dissertation to find a solution to apply agile methodologies to the embedded software development in this company whilst considering their current developing method.

1.2 Objectives

In the context of embedded systems and the company Exatronic, the objectives of this dissertation are:

- Understand agile practices and recommend which are better suited to embedded development.
- Present a solution to change the development process of embedded software projects in Exatronic from the traditional way to a more agile approach.

1.3 Structure

The structure of this dissertation is as follow:

- Chapter 1: As seen, this chapter presents this dissertation's motivation and its objectives.
- Chapter 2: Presentation of the agile methodologies, the practices, some concepts and methods currently used. Description of problems faced when working with embedded systems and some recommended solutions.
- Chapter 3: Description of the requirements and planning the agile intervention.
- Chapter 4: Presentation of process and tools to the recommended solution.
- Chapter 5: Results and Validation of the solution using different examples and projects.

chapter 2

The Agile Value Proposition

Software development methods are used to manage complex projects in order to make developers aware of project priorities and deadlines, by following guidelines and recommended practices. First and foremost, in order to develop a product, we need to know what to develop, that is, we must have goals and follow some guidelines to answer questions such as what the product should do and how. This process is called requirements elicitation [2]. In the mid-1990s, development teams always attempted to obtain all the product requirements before the development process [3]. At that time, the most commonly used development method was the Waterfall Based Model. In this model, it was essential to get requirements first, to be able to specify in greater detail what to develop [3]. However, it was frustrating for developers to obtain all the requirements before the product development. This was because the customers are often unsure of what they really seek to obtain, and only during the project development the requirements become more detailed and fixed [3], [4]. When a first version of a product is released and presented to a customer, they sometimes may not like it and want to change things, which forces the development process to begin again [3]. Due to this began a need to keep the customer informed about the process, asking their opinion about the development process so far. This essentially happened to prevent making any abrupt changes to the product in the future after the initial product demonstrations. Developers unconsciously started to release working versions of a product, and do product tests throughout the development, in order for the client to be better informed. Gradually, this method of developing software was tailored to match the customer's interests [4]. Additionally, the market was under pressure to produce more innovative developmental processes, as well as to increase flexibility between developers and customers in both a quick and economical way [3]. In turn, pressure also began to mount for companies to look for new technologies and new products.

Consequently, the need to find a new method that would meet all these essential changes started to emerge. However, this new method had in fact already begun, but it did not yet have a name. The name chosen was agile methods, a new kind of approach that claimed to be the answer to the problems mentioned above.

2.1 The Agile Approach

Agile methods gained popularity in 2001 when a group of experts from various methodologies of software development established a set of common principles in a document entitled “The Agile Manifesto” [5]. These principles define what being agile really means in an iterative development process [6]. Agility involves using strategies to respond quickly to market changes, create new products, and specify customer requirements not before but during the project development. This methodology encourages teams to be proactive by developing products with only the necessary details to achieve the goal. There is a focus on working software, keeping simple planning with short iterations and early releases, and leaving space for subsequent changes during development [3], [4]. Development teams must be adaptable and prepared for anything unpredictable and, in view of this, communication is the key, not only between customers and developers, but also between the developers themselves [7]. In traditional methodologies, such as Waterfall, development teams try to plan their products with minimal detail, organizing all the steps before starting development, and trying to predict changes that the customer may make [3], [8]. This approach leaves little room for change or the unpredictable.

2.1.1 The Waterfall Process Model

The Waterfall Process Model was introduced in 1970 and accredited to the Walter Royce article [9], [10]. This model is the oldest software development process model, and used to be the method most frequently used by developers [6]. It was successful in some areas, but also had some flaws such as inflexibility, manifested in its difficulty of responding to changes [4]. Consequently, these flaws were observed in some traditional methods too.

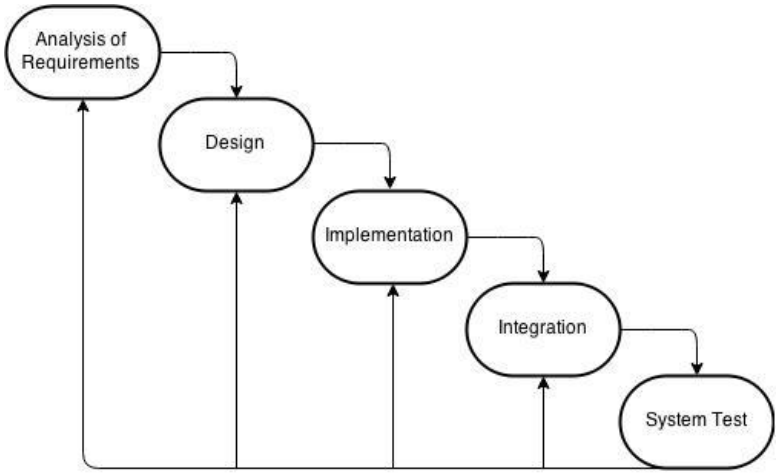


Figure 2.1: Waterfall Process Model, adapted from [9]

The Waterfall Process Model is divided into sequential phases which must be completed in order [9]. The first stage is to gather all the customer requirements, as well as to carry out extensive and detailed planning before beginning the development process [9], [10]. This extensive planning shows waterfall as a secure method because the detailed planning facilitates the estimation of the budget, the development duration and predictable customer changes, among other things [10]. The downsides of this method occur when it is necessary to modify the project design in late stages. This is because the risk for delayed delivery product increases, alongside the consequent increase of development costs [8], [10]. If the modifications cannot be applied, due to, for example, short deadlines, the customer will end up unsatisfied with the product functionalities, or lack of them [8], [10]. Such need to make modifications at late stages may be due to negative customer feedback and/or crucial errors found in the testing stage that forces the alteration of the project design, since testing and feedback are done at the late stages [8], [9], [10]. Currently, researchers associate the Waterfall Model with great risks, failures and low productivity [10]. Royce, however, recommends to do the development cycle twice if the software is being developed for the first time. Although this would increase production time, it would also increase the quality of the product. Royce did hint at iterative development in the article, including some feedback and adaptation, but this was lost in the descriptions that followed this method [11]. The iterative development consists of several sequential iterations. Each iteration corresponds to a goal or part of the project being developed [6]. At the end of each iteration, the aim is to make a stable release of the integrated and tested project. Waterfall methodology was far from being an iterative development.

2.1.2 The Agile Process Model

The Agile Process Model was built based on iterative development [6].Figure 2.2 shows a more flexible model with more space for change during software development. In this process model the requirements, architecture or design are not considered as something static [6]. The model divides the project into smaller mini-projects and for each iteration it is necessary to have something working, a feature [3], [12]. The development team only advances to the next feature after finishing the current mini-project. There are not distinct stages during the development, instead, each feature is taken from start to finish within an iteration [12].

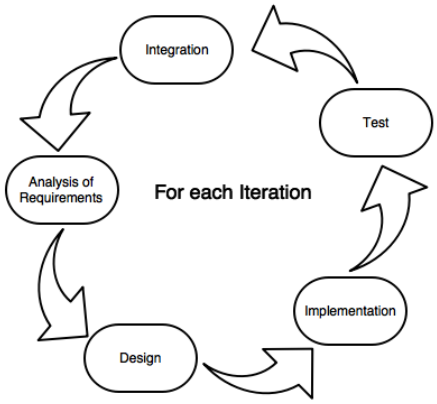


Figure 2.2: Agile Process model, adapted from [6] and [12]

Comparing both process models, agile has an easy rollback during the development process. For example, if an error occurs at implementation due to an aspect of the design stage, it may be necessary to change all the design in the waterfall process. In the agile process model, because these errors are detected sooner, it would only be necessary to modify the design done for the last developed features, not all the design. In the waterfall process model, there are no iterations. Each stage is performed in a detailed way and any project modifications at late stages are very complex and time consuming. However, one criticism of agile processes is that they can be vague. Whereas agile processes only focus on the current iteration, waterfall takes account of everything and can, for example, estimate the product budget.

2.1.3 The Agile Project Team

Agile teams have a team concept different from other methodologies of software development [13]. In the agile method, there are a lot of meetings between team members in which they receive feedback from each other about the work being done [13]. This feedback is given not only by the team manager, but also by other team members in order to know what the current project status is, what has been developed so far, what the difficulties are, as well as to give support to colleagues. The agile teams always try to be cohesive, working together towards a common goal, increasing the members' ability to listen and respond constructively to others' opinions, and to collaborate with and respect one another's work [7], [13]. As a consequence of this, the company has a better work environment [14]. Developers that use agile methods often have great motivation and high job satisfaction [15]. In result, the individual satisfaction increases performance, which in turn increases team performance, and therefore ultimately the performance of the company [13]. By having the customer on-site and having regular meetings, the team is provided with continuous feedback and can easily make any modifications on the project [16]. This increased collaboration between developer and customer means that the customer can always have an idea of how the project is being developed and may improve the requirements and change them according to what they really want, avoiding future errors and confusions that could arise [16].

2.2 The Practices of Agile

So far, some fundamental principles of being agile have been discussed. In order for a software development method to be agile, it needs to adhere to these previously mentioned principles. The practitioners who wrote the "Agile Manifesto" proposed particular agile development methods and specified in detail which characteristics may lead to better development, as well as which practices [5], [14]. These practices vary according to what is intended in each project and the type of product that is to be developed. After an analysis of the practices of some the most popular agile methods, for this section we choose the most notorious and distinctive for two reasons. On one hand, the practices which are considered essential for a method to be agile. This is based firstly on practices that differentiate this method from traditional ones and define it as being agile, and secondly on the most applied practices with positive results in articles mentioned throughout this dissertation [14], [16]–[18]. On the other

hand, I selected additional practices which can be used according to the preferences of the development team. These practices are more specific, and less used than the others [14], [16]–[18]. All these practices will be described, as follows, beginning with the essential ones.

Coding Standards [19], [20]**:** The code should follow coding standards in order to make the code perceptible for the whole team to read and refactor. Hence, make someone's code available to all team developers, without wasting time on understanding it.

Continuous Integration (CI) [21]**:** In continuous integration, developers frequently integrate their daily work on a central repository in order to ensure that system releases are fully-working and bug-free. Having continuous integration improves quality of code and prevents risks because it anticipates possible future errors that would otherwise be invisible without the integration of all the product code.

Customer Feedback [3], [14], [16], [22]**:** The agile development team cannot consider the initial requirements as definitive, since they can change during the course of the project. This happens because the customer may alter the features' priorities, and developers must be prepared to adapt and react quickly to changes in order to satisfy the customer. Customer feedback involves frequent meetings between the customer and the developers. These meetings usually happen during the demonstration of system releases, after each project's iteration.

Frequent releases [21]: This practice, together with test driven development and continuous integration, aims to ensure that working software is always available.

Product backlog [22]**:** This consists of a priority list of functionalities not yet implemented in the product. The customer makes a brief description of the high level functionalities that he wants, which are written in the product backlog. During project development, the items can be changed (added, removed or given a new priority), which makes the difference with the typical list of traditional methods. It is important to have a list of what is intend to be developed because these priorities force the team to be more focused on what is really essential.

Refactoring [19]**:** Refactoring is about restructuration of the code by removing duplicated code and simplifying or adding flexibility, without changing its behavior. This practice attempts to facilitate modifications of the code, for example, on adding new functionalities. If the code is simple, the time and effort to do it will be just enough.

Simple design [21]**:** Keeping the design simple is about not implementing features that are not requested by the client. This technique is known as YAGNI, for "You Aren't Gonna Need It". The developer may think that a more complex design may be useful or requested by

the customer in the future, planning what might change during the project development, which is generally a traditional thought. However, the agile principles suggests leaving it simple and not make the system unnecessarily complex. Thus, there is space for unpredictable changes and any future complex problems can be prevented.

Test driven development (TDD) [21], [23]**:** The principle of TDD is firstly to think about the code to be written, then write tests, and only afterwards write the code. TDD encourages the use of automated tests to be done after any changes in the program, ensuring that these changes will not negatively affect the existing code. The functional tests, which happen in the late stage of the agile process model, must continue to be done after the completion of each new feature. The resulting benefits are a more maintainable code, a higher probability of a bug-free product and a lower probability of finding complex errors requiring time-consuming solutions.

These eight practices can clearly describe where agile methodologies aim to make a difference, compared to traditional methods. Other, more specific agile practices will now be described. The adaptation of the next practices depends on what the company objective is: whether it wants to be agile in a more management way or whether it is prepared to embrace different ways to work with.

40-Hour Week [14], [20]**:** Teams usually tend to strive too much to delivery projects on the due date. This can result in a lack of ideas, tiredness, rash decisions, and overcommitted deliverables. This technique encourages the use of only 40 hours of weekly programming. If the team want to do more work on a week, the following one must be normal, without the same amount of previous effort. This technique is not to be taken literally, but rather to control the number of additional working hours.

Collective Ownership [19]–[21]**:** This technique conveys a feeling that everyone is responsible for the code produced on the project. This practice does not distinguish who did what. All the team owns the code, and all developers can change it to improve the implementation by fixing any error or contributing with new ideas. It is intended that developers learn other parts of the system and feel responsible for the quality of the code.

Daily meetings [21]**:** The purpose of daily meetings is to put the whole team abreast of what was done until the meeting, what will be done after, as well as any team difficulties. There may also be feedback on the current status of the project or discussion of ideas to solve problems that the development team may have. However, the duration of daily meetings is usually short to not become exhaustive or an unnecessary waste of time.

Pair Programing [19]–[21]**:** This consists of two programmers sharing a single machine, both having an important role. Whoever has the keyboard is called "driver", because it is their responsibility to consider the best way to implement methods. The other programmer's responsibility is to think strategically about what is being developed. Both are actively involved in the development and they switch roles throughout the process. Having pair programing increases the chance of finding and avoiding early errors and makes both developers learn new programming skills with each other.

Sprints [3]**:** The definition of sprint intends to give the idea of an iteration, a certain amount of time where some product features must be developed. Sprints can usually last between one week and one month. On every sprint, developers must implement functionalities that were prioritized in the sprint backlog in the sprint planning meeting. Inside the concept of sprint, there are four more practices to explore:

Sprint backlog [22]**:** A priority list of functionalities taken from the product backlog, which must be developed during the sprint. The backlog belongs only to the development team. In other words, only the development team can modify the list during a sprint. Customers can do it only in exceptional situations.

Sprint planning meeting [24]**:** The development team defines new priorities for the next sprint. Only the development team can say what they can or cannot develop in the next iteration, creating or redefining the sprint backlog with new objectives. This meeting is essential for the purpose of letting the entire development team know what is really necessary to make on the next iteration.

Sprint review meeting [24]**:** An informal meeting that involve the customer and the team leader to demonstrate the product functionalities already done. The objective is to get feedback about the last iteration development and if necessary adjust the product backlog with the customer.

Sprint retrospective meeting [24]**:** The development team tries to understand what went well in last iteration, and proposes improvements for the next one. The main topics discussed can be agile practices, tools used or the team itself, for example.

User Stories [20]**:** User stories are like customer requirements, the difference is the level of detail given. They are less detailed than the normal requirements. Developers go to the customer and get a detailed description of the requirement only when they have to implement the feature of that user story.

These practices are oriented to the project management, essentially the meetings and the 40 hour rule. The pair programing, for example, it is a practice that must be well considered depending on the type of team and product being developed.

2.2.1 Selected Methods Survey

It has been mentioned, across this chapter, the existence of agile methodologies and agile practices. Yet, these methodologies will now be described and practices of each method will be enumerated. Some of the lesser known methods will be simply described and followed by two of the most known agile methods.

Dynamic software development method [25]: This method is based on nine principles: Addressing current business needs; Active user involvement; Team must be empowered to make decisions; Focus on frequent delivery; Integrated testing; Stakeholder collaboration. It is suited for software development that places a high importance on the user interface or usability aspects of products. The projects are divided in three phases (pre-project, project life-cycle and post project) and prioritized using MoSCoW Rules *(musts, shoulds, coulds* and *won't haves)* in order to deliver the product on time.

Crystal methodologies [26]: A family of agile methodologies that vary based on the size and complexity of the project. Crystal Clear, Crystal Yellow and Crystal Orange are some examples. Each color represents the project size and criticality. Size is defined as the number of people involved in a project and criticality is defined as the potential for the system to cause damage. They are considered and described as lightweight and easy to adapt.

Feature-driven development [27]: The domain model is one of the central artifacts of FDD. While most agile methods start with a set of principles, the center of FDD is the domain model. The iteration development is based on features. This practice relies on specific team roles, tending to move away from the practice of collective ownership since each role has its own responsibilities.

Lean software development [28]: Its principles are based on the Lean Enterprise movement and the practices of companies like Toyota. The main principles are eliminating waste, amplifying learning, deciding as late as possible, delivering as fast as possible, empowering the team, building integrity and seeing the whole picture. This method is similar to Scrum (it will be described later), it focuses much more on the project management aspects of software development. However, requirements are measured based on their impact to the

business, and must be specifically defined in a clear and complete way. Incomplete requirements are filtered out during the analysis of requirements' phase.

Among many agile methodologies, there are two that stand out for being the most studied and well known. These are Extreme Programing (XP) and Scrum [14], [16], [17], [29]. This does not mean that using one makes it impossible to use the other [30]. It is possible to use both according to whichever best fits the project. While Scrum uses practices related to management and control, XP focuses on programming practices [30]. Figure 2.3 presents the results of an empirical study, conducted at Microsoft, and refers to which agile method developers use. The results show that Scrum is the method most used. For developers who did not know which agile method they use, they were asked to specify the method used, and the study concluded that most of them were variants of Scrum.

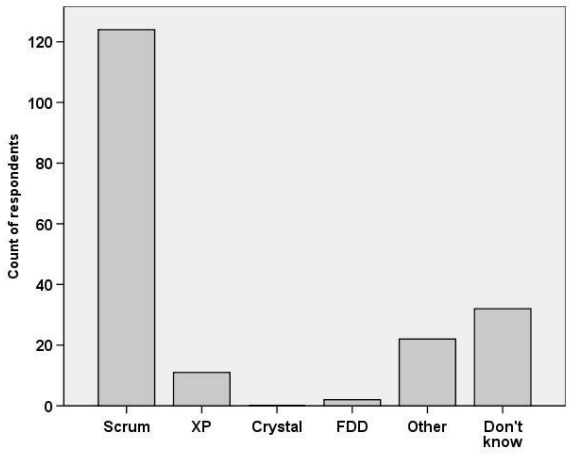


Figure 2.3: Different Agile Methodologies, presented in [16]

Extreme Programming

Extreme programing (XP), originally proposed by Kent Beck, is another way to develop software. In this method, customers write user stories and, as previously mentioned, when the developers have to implement a user story, they go to the customer and get a detailed description of the requirement. It can be concluded that the availability of the customer is a requirement for the use of XP [31]. The customer is part of the team for all phases of XP, thus, communication with the customer (preferably face-to-face) is required [20].

After defining user stories, a release plan is made containing the user stories that will be implemented in each system release and the date of those releases [20], [31]. This plan is defined in a release planning meeting, where the customer can specify which implementations should be implemented in the first place, according to the costumer's interests [20], [31].

For each iteration, an iteration planning meeting is held where developers take the user stories and break them into tasks [20], [31]. At the end of each iteration, the system release must be tested and functional, in order to be shown to the customer [20], [31].

XP is most known for its technical practices, especially test driven development, pair programming and collective ownership [19]. Among the practices mentioned in the section 2.2, characteristics of XP are continuous integration, simple design, coding standards and 40-hour week [17], [19], [20].

Scrum

Scrum is a way to develop software in small pieces, with each piece building upon previously created pieces. A scrum team is composed of the product owner, the development team and the scrum master. The product owner is typically a stakeholder who determines what needs to be built by creating a prioritized wish list called a product backlog [22]. The development team is responsible for delivering the items defined on the product backlog. The scrum master is in charge of solving problems and keeping the team focused on its goal [6].

In scrum, iterations are named sprints and usually lasts for one month or less [32]. For each sprint, the scrum team pulls a small chunk from the top of the product backlog, puts it in a sprint backlog, and then develops these items. The sprint backlog is made during a planning meeting, and after each sprint the development ends with a retrospective and review meeting [22]. Furthermore, there are daily meetings where the team members coordinate work [22]. As the next sprint begins, the team takes another chunk of the product backlog created by the product owner and begins to work again. The cycle is repeated until the items in the product backlog has been done. Among the practices mentioned in the section 2.2, characteristics of scrum are daily meetings, customer feedback, product/sprint backlog, and sprint meetings are Scrum characteristics [22].

2.2.2 Accessing the Impact of Agile Practices

Survey results in the integration of agile methods in development team can change according to which method they are using, their experience and the size of the team [14]. Agile methodology is considered something new and, therefore, developers argue that an introduction is necessary before using it [14]. In relation to scaling this methods to larger teams, most people agree that these methods are only appropriate for small teams where communication is easier [16]. In general, companies which adopt an agile method are satisfied with it, developers are more cohesive and happier with the job, regardless the product quality being superior or equal compared to the old method [14].

In general there are good and bad experiences with agile practices. For example, having customer feedback is something that requires hard work and customer concentration [14]. The customer may have to learn some things about being agile and developers must spend time on that and/or the customer may be unable to attend many meetings [16]. But on the other hand, some clients praise the continuing involvement with the project because this made them to have control over the development process [14], [18]. For developers, this involvement is important to make early modifications on the project, improving the quality of products [14]. The fact that the customer has some control in the project makes the product vulnerable to changes [3], [6]. There is a competitive advantage on this. The ability to produce new and different products for the market is bigger because the customer is aware of what is happening in the market and rapidly changes this requirements, leading the team to develop more innovative products [18].

As mentioned throughout this chapter, in agile methodologies communication is key. Daily meetings is an example of a practice which encourages it. The higher the communication is between team members, the better the awareness of the project status is. Communication favors aid among team members, making the discovery of errors much easier to resolve [16]. Agile always tries to encourage the communication, and consequently, the individual motivation and satisfaction grows. As a result, team performance is better, causing the products to be delivery on time [7], [18]. However, the team meetings, which generate communication, are often only used to report to the team manager what each developer did that day, in particularly, the daily meetings [14], [16]. This could be a management problem. Many team managers do not have the skills necessary to manage an agile team [16]. If a manager does not know how to apply good practices of agile methods, then the project may be at risk. The same problem happens in less experienced teams because they may not have the practical

domain to adapt some agile practices as the experienced developers have or they may not be so flexible to quickly adapt to changes [14]. Clearly, agile development is not suitable for everyone. Indeed, there are teams that cannot adapt to agile methods, even with experienced developers [16]. Anyway, there are agile practices that allow to have a better management to the development team not lose track of the real goal, and delivery the product on time, but it must be well oriented by the team manager.

Finally, many developers compared the waterfall process model to the agile one, saying that there is a big disadvantage in changing from one to another when it comes to planning the project. It is hard to predict the budget and the time wasted on the product. Agile projects do not have a good detailed planning and the project development may be a little bit vague, as mentioned before. The second concern in changing is the difficulty with introducing an agile method in a complex organization, because complex projects have much more teams and even more elements. The communication between everyone involved is more difficult and sometimes impossible, making coordination very difficult [14], [16].

2.3 Software Engineering Practices for Embedded Systems

An embedded system is a combination of software and hardware, a system that has software embedded into hardware [33]. Embedded means built into the system, specifically designed for a particular function which may be a part of a bigger system [33]. Usually, embedded systems do not need a full operating system, but like any computer, it has an input and output [33]. They are typically referred as reactive systems because embedded systems produce different results on the output according to the changes in the input [33]. For example, an elevator controller is embedded in an elevator and tell him to move to different floors based on the input. Traffic lights, automobiles, wireless handsets, mobile phones, vending machines, medical equipment, toys, airplanes, clock radios, ATMs, network routers, MP3 players and video game consoles are examples of possible hosts of an embedded system. If we look around, embedded systems are found everywhere. Embedded computer systems are diverse and control many aspects of modern life. In our house, we may have around 50 embedded systems [34]. They are built to perform tasks more reliably using simple and cheaper hardware.

The development of embedded systems is different than software applications, like a website development, and therefore require a different approach [35]. On this section, it is

presented a review of engineering process for embedded systems, focusing on special requirements.

2.3.1 Specific Challenges

Develop embedded software it is a hard challenge for those who are accustomed to application development [35], [36]. They have unique characteristics that differentiate it from other software developments [35]. Embedded Systems are quite complex, if a computer has an error on the software, once discovered, a software patch can easily fix the problem, or a simple reboot. On the other hand, embedded systems are often programmed once and the software cannot be fixed, unlike the computer software [37]. Even if it is possible to correct the problem, frequently the process is too complex for the user. Software applications such as music player, or pdf viewer, may fail, but the consequences are the loss of data or the music stops playing. In some embedded systems, an error results in a product that no longer works and maybe needs replacement [38]. Some embedded systems do not allow unreliability, errors are unacceptable, and the product is not allowed to fail [39]. Some systems can cause serious damage on valuable equipment or threaten human life, if they fail (e.g. in spaceflight and automotive systems) [39].

Reliability, maintainability, efficiency, safety, security, and cost are some of important characteristics to have in mind when developing embedded systems [33]. Reliability and maintainability, in some ways, are complementary. Some products require technical support to respond to failures, or to make routine system checkups [33]. With higher product reliability, technical support will be less needed[33]. Having more reliable systems means having a less corrective system [33].

Another challenge that is an increasing concern in embedded systems is security [40]. These systems are ever more getting involved in telecommunications and network industries, and therefore in transfer of secure data, for example, through public networks that need protection from unauthorized access [40]. Security requirements must be addresses in order to meet the variety of challenging security [40].

There are also embedded systems that must meet hard real-time constraints and some of them can cause catastrophic consequences [33]. For example, a pacemaker is a real-time system which it reacts to an input within a specific time period [33]. If the pacemaker has a delay, it may result in death.

That said, agile is more directed to less planned systems, not critical ones, because these systems need a very detailed plan before the development. Non critical embedded

systems are dedicated to specific applications and are normally compact systems with resource constraints, such as limited memory, simple processors and low power. The software must respond to any hardware limitations. However, initially, developers may not have the target machine to test these limitations and may need to use alternative testing to overcome this problem. This happens because developing embedded systems is about developing both hardware and software concurrently, and the hardware may not be available.

All these challenges make embedded software development different than the application development. The lack of methodologies and tools to support specific embedded requirements makes the development even more difficult [41].

2.3.2 Embedded Process Model

Typically, embedded systems have a workflow like the one in Figure 2.4. This process model attempts to satisfy the requirements of hardware and software to integrate and develop the final product. The model starts with the specification of functional requirements, with simple details of the implementation and non-functional requirements such as memory, power or cost. In other words, performance constraints [42], [43]. Since embedded systems are SW/HW architecture, after the architecture stage, there are two developments in parallel [44]. On the hardware path, there is the design creation where some developers use hardware description language (HDL) [45]. VHDL[46] is an example of HDL. After implementation, tests are done to ensure some performance constraints applied [45]. On the other path, the first step is the software design which can be done using UML, for example [47], [48]. After the implementation, tests are done to finally have a prototype to integrate with the one made on the hardware path. Then, software and hardware implementations must be integrated, tested and searched for defect, to ensure that they meet the specifications [42], [45]. If any errors are found, the team may have to re-design the hardware or software part, depending on the type of error [45]. The last stage focuses on the maintenance and support of the developed product. The next sections intend to study the application of agile methodologies on the software path, taking hardware limitations into account.

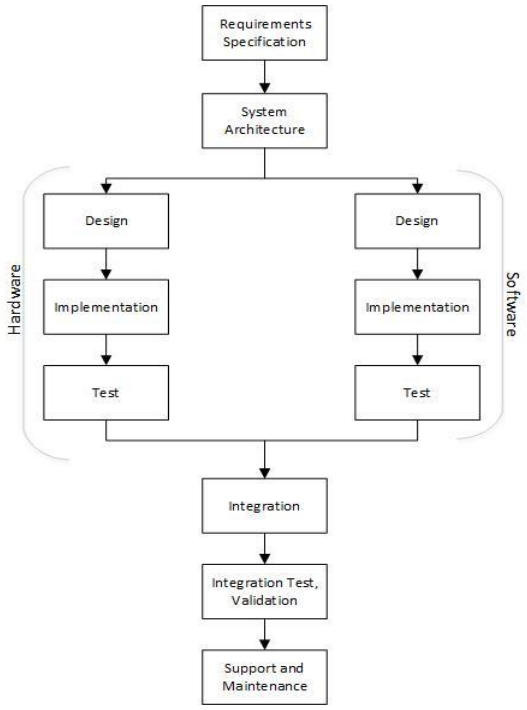


Figure 2.4 Typical embedded system process model, adapted from [44]

2.4 Agile Practices Adoption in Embedded Systems Development

Embedded systems are becoming increasingly smaller and cheaper which in turn has led to a great demand on the market [49]. They are cost-effective, which makes them used more frequently in large and complex projects.

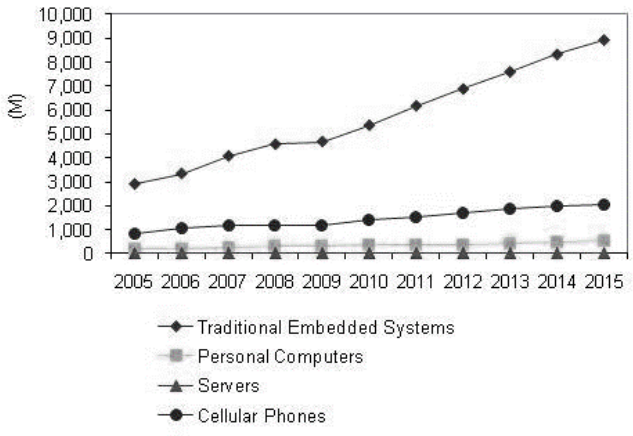


Figure 2.5: Embedded systems and other mainstream system shipments, presented in [74]

The growth visible in Figure 2.5 shows that companies are increasingly focusing on developing embedded systems. They have to develop cheap quality products within a well-defined time limit to meet time-to-market goals[43]. However, this has been challenging for companies [43]. The pressure to deliver a quality product on time is high because the state of the market is unpredictable. Another company can release the same or better product at any moment, and the customer may want to change requirements in middle of development because of this. Late changes, in the development of embedded systems can be critical, since these systems depend not only on the software side but also on the hardware integration [50], [51]. Changes in hardware architecture at advanced stages can create more costs and increase the development time. The consequences of this on the implementation of software is not known for certain [50], [51]. Having space for change, using agile methodologies which not detail the architecture so early on, makes possible to quickly respond to requirement changes and future problems.

In addition, independent and parallel development of the hardware and software, as seen in Figure 2.4, may result in a very complicated integration with too much risk for the product delivery time [50]. If a design error is discovered during the development, either the software developer is restricted to the hardware architecture and it must be modified, or the hardware developer needs to, for example, create a new component. If the development team is restricted to the hardware architecture, they have to develop the software taking into account the hardware design and his limitations. Companies that have all these problems may need to adapt their normal lifecycle to this new complexity. They must use appropriate methodologies to be able to follow the market [43].

Agile methodology claims to resolve some of these problems, but it is necessary to carefully study the agile methods, and try to adapt them, to be well applied in the development of embedded software [41].

2.4.1 Literature Review

This section will analyze the most used agile practices according to a study on European companies [17], by using study results and recommendations of applying agile methods in embedded projects. The study results are presented in a table in the Annex A. For each work the table contains a set of adopted practices and its results.

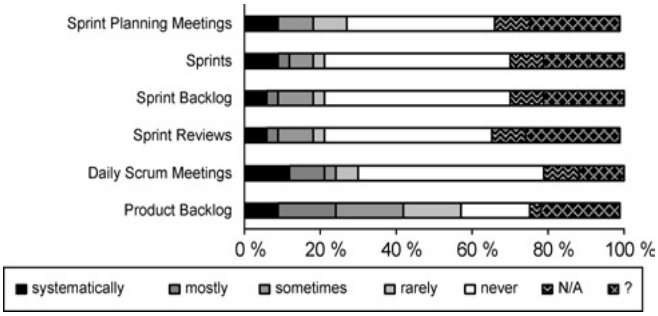


Figure 2.6 Use of Scrum practices in European embedded software , presented in [17]

Figure 2.6 shows the results of a survey on the actual use of Scrum in European embedded software development organizations. The most used practice is the product backlog. According to the study having a prioritized list can be something universal, not necessarily a Scrum thing since it is always used anyway.

Daily meetings is the second most used Scrum practice. They were important in the initial phase of a project in [35] because they gave really good feedback, but team members quickly got tired of these meetings because they turned to be only with what each team member did at the present. This is also supported by the studies done in [16], where developers felt uncomfortable stating their progress every single day. However, it was seen as benefit too, by making team members aware of what the others members were working on, and promoting the early discovery and handling of development issues.

According to the survey, Sprint planning meetings is the third practice most used in european companies. The article in [35] states that, before applying this practice it was hard to understand how far from the end the tasks really were, which led to a lot of unfinished tasks for a long period.

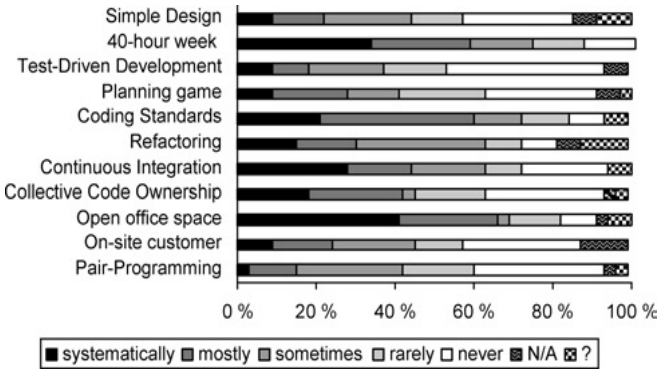


Figure 2.7: Use of XP practices in European embedded software, presented in [17]

While Figure 2.6 is related to Scrum practices, Figure 2.7 is related to the XP method. The practices 40 hours for week, coding standards and open office space are the most applied techniques, but these ones are more simpler and easy to adapt.

The next practices most used are refactoring and continuous integration. According to [52], refactoring work is better to do in the early project because it can fix a lot of bugs which would otherwise take a long time to debug. In [35] developers were already applying it without having a name for it, but after knowing what refactoring was, they always looked for chances to use it. The same study states that refactoring is slowly improving the quality of their legacy code. For [49], this continuous process of refactoring cut out duplicated code, reduced the amount of system functions, and improved the system performance.

Regarding continuous integration, in [52] this practice is used for each component, but the full system integration is done only when releasing the product, because it is a complex system and they need external test equipment to do it. In [53], this technique allowed them to identify areas of functionally which had been unintentionally left public and been used by other components.

Fourthly in the survey comes the collective ownership. In [52], before this practice, either developers maintained the code or spent time teaching it to someone else. However, they only found this practice appropriate for single teams, not crossing teams. Collective ownership was applied in [35] too, but was initially complicated because team members did not felt secure when other developers improved/changed their code, but in the end they accepted. Studies made in [17] concluded that collective ownership is one of the most used and appreciated XP practices, with no negative experiences.

Pair programing is the least used systematically. This practice was seen as something efficient in the first stages of the project in [35], because the practice finds defects sooner than testing. But after that was just inefficient. Now the team just do pair programming during the design phase and initial code phase. In [52], after using pair programming, the team concluded that the required quality code level was achieved earlier and the pair developers learn a lot from developing together, which is also supported in [53]. On the other hand, it was found to be unsuitable for simple or well-understood problems, which could be fixed as quickly as a single developer could type. Studies in [53] add that with pair programming the objectives for the pair was achieved with lower time and lower bugs than a single team member.

On-site costumer is one of the least used practices in conjunction with test-driven development, perhaps due to the complexity, the big culture change and the limited financial resources for applying them. In [53] the flexibility and involvement of customers on daily basis changed the team focus to the customer's interests. They were focused on helping to meet the

requirements of their customers. In [35] they do not have an external costumer to negotiate with, however they made the hardware design team their customers. In [52] developers did not use this practice, because they do not have specific customers.

In TDD, according to [35], it forced developers to think about the code that they would do before coding. In [52], developers could better understand the required functionalities from a client's point of view. The studies made in [16] report that test-driven was one of the factors that improves the high quality of the produced code.

2.4.2 Agile Practices that Can Lead to Success

Before a team starts thinking about applying agile methods, it is necessary to organize what is important to apply and what is necessary to change. This is named process tailoring, where the team is organized in order to use the best practices for agile project development [50]. This process is fundamental and it is used in this dissertation by combining agile practices from different development methodologies, whilst trying to meet the needs of embedded projects.

This section will identify what the key factors to adopt agile in embedded could be, based on the information gathered from the literature review, in the previous section and resumed in the table in Annex A. Previously mentioned challenges will then be discussed. The first practices to be mentioned refer to the way of programing code.

Coding Standards: Usually companies require developers to use coding standards. It is worth mentioning that if someone wants to, for example, refactor or fix code already done by other developer, the time spent on understanding the code is wasted time. With coding standards and code improved/refactored it is easier and faster to understand and modify code.

Continuous Integration: Developers have to integrate new changes on their workstation before they can commit to the repository, preventing future errors that if only detected at the end of the project it would delay the delivery. This practice ensures that the embedded systems constraints are met, when integrated with what has been developed so far. This can be done using a continuous integration server which will ensure the build of the project as well the usage of tools to analyze the code. For example, every build made on the CI server, can run a set of tests and, if errors are found, the developers will be immediately warned. When Test driven development and Continuous integration are used together, the product has less

probability to have errors and integration problems. When is almost certain that errors will be found early, it will be cheaper to correct if it involves hardware components.

Refactoring: Refactoring is done at the end of each functionality to improve the structure of code, making it more flexible and consistent. During the creation of automated tests, when a test passes it is a good opportunity to clean the code and verify if the test does not fail. This technique can eliminate some time wasted on debugging code, if done from the beginning. The quality of a product will improve with the elimination of unused and unnecessary code along with the reduced size. Refactoring may be thought as non-priority thing to do, but it is almost certain that it will help to release quality products.

Test-Driven Development: Making tests before implementing features helps developers to understand more about the functionalities and what the customer really wants. After developing a functionality, it is necessary to do more testing including both functional and integration tests. It is essential to do this during the project development. Running automated tests will save time that would be wasted on late debugging and manual testing. Continuous testing helps to anticipate future decisions of changing the hardware design and to prevent the use of software solutions to resolve problems that cannot be resolved due costs and limitations. Additionally, embedded systems have performance limitations that need to be taken into account and the early and continuous testing of hardware characteristics is essential to ensure these non-functional requirements. However, in the first stages of the development process, the hardware may not be available. The solution is to use evaluation hardware (evaluation boards) and the development system, they may be the only way to ensure early tests. Anticipating future changes leads to a fast development to meet the time-to-market goals.

In order to carry out the next recommended practices, the team should be organized. It is necessary to assign roles and responsibilities, determining who is best suited to do certain work. It is also fundamental to have someone responsible to make sure that the recommended practices will be well applied. The next described practices will identify the key factors of management practices.

Customer Feedback: By having customer feedback, it is possible to know what the customer actually wants, enabling developers to focus on what they really must implement. It is possible to get this feedback through the iteration planning meetings, where the customer sees a demonstration of the product functionalities developed so far, giving their opinion and explaining the next necessary features to develop in the next iteration. Customers can ensure,

for example, whether the hardware requirements are being fulfilled, and developers can prevent late changes.

Daily meetings: This technique makes knowledge transfer and sharing both possible and easy, which is essential for agile teams. The hardware and software teams can discuss the design in these meetings to ensure a development of dependency, not independency. Complex HW and SW integration errors may lead to the redesign of the architecture, something that can delay the development of the product considerably. However, there are reports that daily meetings only covered what each team member had done at that point in time. Therefore, if meetings get to this point, they must be brief, not taking too much time from the development, or they must be stopped. These meeting are very useful in the beginning of the development and the team must decide whether to continue meetings based on their usefulness.

Sprints, Sprint planning meeting and sprint/product Backlog: These Scrum practices make developers focused on project objectives, which should be achieved in each iteration. Scrum is the most agile methodology used and if combining these practices with others, the development teams may have a successful methodology. Product and sprint backlog make developers aware of what they have done and what they still must do, without losing track of the project. Sprints allow developers to commit themselves to complete their tasks on time, while having flexibility to any changes. To do so, the sprint planning meeting is essential to plan what features are necessary to develop.

User Stories: User stories can be very useful because encourages more communication with the costumer. For each iteration it is necessary to talk with the costumer to specify the respective features. This leaves no space for misunderstandings with regards to what the customer really wants.

Some practices can only be advantageous if well applied, others must be well-considered. It depends on the type of product being developed and on the developers. The next final practices give an example of techniques that only work in some projects and teams.

Collective ownership: Using collective ownership makes everyone feel responsible for the code developed for the project. All the team developers can make changes to anyone else's if that person is busy doing, for example, another functionality. The time saved using this technique can be a step to shorten the product development. However, some teams may not be comfortable with this, and for others it can be irrelevant because developers may not have the knowledge to do other things.

Frequent Releases: This practice makes sense when having the hardware available since the beginning of the project. Usually, hardware is only available almost at the final stage and the testing and integration are done in evaluating hardware, which prevents early releases to the market. Frequent releases can be useful when the product development takes longer than normal and it is necessary to release the product to market with the main features, as well to get a clear feedback from the customer [21].

Pair programming: This is a technique that offers positive and negative aspects. With companies that have novice engineers, it can be applied because they can learn the system and the process while experienced engineers answer questions that they might not have previously considered. However, there may come a point where it is not efficient. Wasting two developers to do work that one developer could do is wasting time and this goes against agile principles. This practice is not advisable in small size teams with projects with simple tasks that can be done by one developer.

2.5 Summary

Technology is growing quickly, competition is high and it is safe to say that an idea today may be outdated tomorrow. Companies want to be one step ahead, looking for solutions that lead to success. The agile methodology claims to be a response to the instability of the market, using methods and practices that attempt to respond quickly to changes. Nevertheless, for these agile methods be adopted, it is necessary to use non-traditional practices, that is, a change is required in how software is developed. It is not easy to introduce agile methods overnight when experienced developers are still accustomed to a more traditional thought. Agile methods try to adress the instability and variation of requirements by using several practices and by not requiring a detailed specification in the requirements stage. Above all things, communication is the key to being agile, by having continuous feedback from both the customer and the software development team. Regarding the methods themselves, Scrum and XP are the methods most adopted by companies, and also the most analyzed in articles. Whereas XP is focused on programming, Scrum is focused on the control and management of the team. It has been verified that Scrum is not only the most used method but it is also the easiest to implement, given that it does not require radical culture changes as in XP.

To summarize, practices in embedded environments are recommended, although in an abstract way. Each development team is different as is each project. It is up to the team to select and tailor the practices that they think that are best suited to them and to the kind of project. These practices will not make a business succeed overnight. A good introduction and implementation of the agile methodology and its practices is needed. It may take time to notice the changes, but in the long-term results are expected to be positive.

chapter 3

Proposal for Agile Practices Adoption

After studying what agile practices are and which best suits for embedded development, is time to look to the company, Exatronic, analyze what they do, how they develop and how this dissertation can step in and try to introduce a better development. To help with this, Exatronic provided a finished project where it was made some unit tests.

3.1 Requirements from the target company context

This work aims at finding valuable and practical contributions for embedded system development. We depart from the context of a real company, Exatronic, which has kindly agreed to share its development practices. Exatronic develops firmware and hardware, either for automotive, electronics or medical devices. The development teams are small and the products are not too broad in scope. Exatronic can develop hardware and firmware for the same product, but can also develop each one separately. For embedded development, they use Atmel microprocessors and the Atmel Studio for the firmware. The software development tool used is a free integrated development environment (IDE) for developing and debugging Atmel ARM Cortex processor-based and Atmel AVR microcontroller applications [54]. Atmel Studio runs only on Windows operating systems. Normally the embedded development occurs without automated tests and the code integration is done using a SVN repository. Developers have a good relationship between them, both hardware and software teams, encouraging informal communication and collaboration.

Taking into consideration that we aim at providing practical recommendations for a specific company, the range of feasible practices should be analyzed. In section 2.4.2 some

agile practices were recommended and divided into three categories: code practices, management practices, and other specific practices. Now, they will be tailored to be implemented in the project provided by Exatronic. The objective is to implement them in the project to show that is possible to apply some agile methods, taking into account the company development tools. The Table 3.1 summarizes which practices could be possible to implement in the project and include in this dissertation. A more detailed description is presented below.

Management practices

These practices need to be used since the beginning of a new project. The Exatronic project is already completed, so it is not possible to apply management practices.

Code practices

The code techniques are related to the way the development code is done and, therefore, some can be implemented in the project.

Coding Standards

This is not a new practice to the company. These standards were respected during the development of the finished project, that is why these practice will be not included on this dissertation.

Continuous Integration

This practice is not used in the company. The integration process is done with a central repository and SVN. Using the Exatronic project, it is possible to simulate a continuous integration environment and generate results from the static analysis tools and the project unit tests.

Refactoring

This practice needs to be implemented since the beginning of the project development. Anyway, having the static analysis tools and doing test driven development it will be possible to get results that will encourage the use of this practice and, therefore, refactoring will be included, but only partially.

Test Driven Development

TDD is not used in the company, it was only made unit tests on the project provided for this dissertation. This is why TDD was included, in order to simulate a TDD environment to allow the use of the project unit tests.

Other Specific Practices

The practices from this group may not be useful in some embedded companies. For the finished project, some cannot be implemented as described below.

Collective Ownership

To adapt this practice, it is necessary to have all the production code visible to all developers, as well the results from CI server, including the test results. It is also necessary that all developers have the necessary knowledge to feel comfortable to modify code from others. This is not possible to implement because there is no development team, however is possible to make available all the information about the project, for example, the production code from each developer, which tests failed, who committed code with bugs to the repository, among others. This information comes from the CI and TDD practices, which will be implemented, that is why collective ownership will be partially included.

Pair Programming

This practice needs to be applied during a project development and needs also, at least, two developers to do the pair, this is not possible to have with a finished project. Pair programing will not be included on this dissertation.

Frequent Releases

It is impossible to have frequent releases with a finished project, it can be only done during a project development.

	Will be Implemented?	Reason
Coding Standards	No	The company has their own coding standards and they were respected during the development of the project.
Continuous Integration	Yes	It is possible to simulate a continuous integration environment using the Exatronic project.
Refactoring	Partially	CI and TDD can give results that will encourage the use of this practice.
Test Driven Development	Yes	It is possible to simulate a TDD environment to allow the use of the unit tests from the Exatronic project.
Frequent Releases	No	It is impossible to have frequent releases with a finished project.
Pair Programing	No	Needs to be applied during a project development and needs also, at least, two developers to do the pair .
Collective ownership	Partially	Is possible to make available all the information about the project and share code between developers.
Management practices	No	The management practices cannot be applied on a finished project.

Table 3.1: Motivation to use the recommended practices

After the tailoring process, is possible to conclude that TDD and CI are the only practices that can be implemented on the Exatronic project, and therefore, they will be included in this dissertation. Refactoring and collective ownership will be partially implemented because TDD and CI creates opportunities to apply them, but it will be not given the full advantage of using them.

In this dissertation it will be suggested a new way to develop embedded software. The objective is to present a solution to allow the adaptation of TDD an CI in the embedded software development, taking into account the actual method of development in Exatronic. In the company, the development is being made mostly using traditional methods, but with some curiosity and attempts to use testing driven development. In order to validate the solution that will be presented, Exatronic has provided an embedded project with some unit tests. Now, it is necessary to plan an intervention on the Exatronic development method, taking into account the way how they develop embedded software.

3.2 Planning the Agile Intervention

This dissertation will focus on presenting a strategy and setup to allow TDD and CI to the embedded software development in Exatronic.

In the first place it is necessary to describe the actual method for developing embedded software in Exatronic (see Figure 3.1). Developers are using Atmel Studio to write code and then they commit and update modifications using the SVN repository.

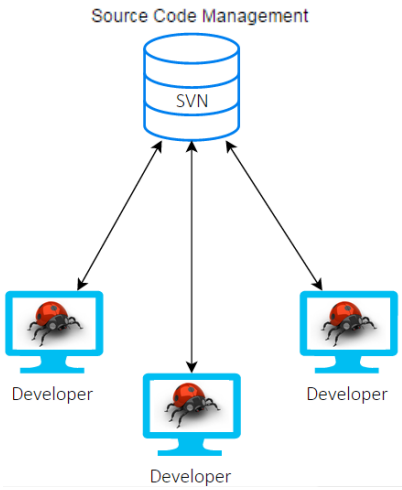


Figure 3.1: Develop and merge workflow in Exatronic

The plan for agile intervention starts by adapting Exatronic workflow to allow TDD. For this, we need a setup that makes it easy and practical to run unit tests in the developer's workstation. For this we propose CppUTest, an open source C/C++ based xUnit framework, written in C++ and used for testing C/C++ projects, especially for embedded ones [55]. James Greening, one of the agile manifesto creators, is one of the CppUTest founders and maintainers [55], [56]. These two factors are the main reasons for choosing this framework for unit testing.

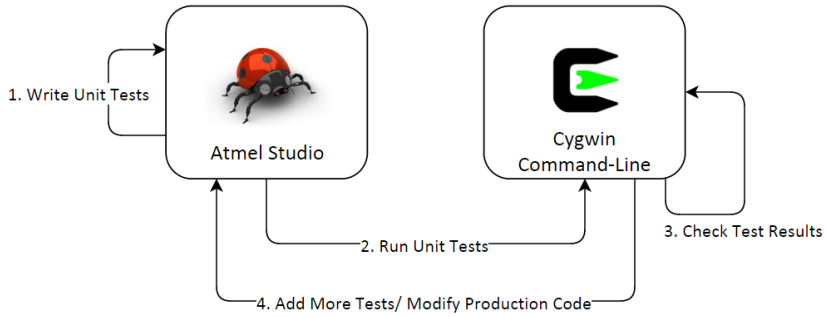


Figure 3.2: TDD workflow used in the Exatronic project provided

The sample project provided from Exatronic uses the CppUTest framework which only works in POSIX environments and can be emulated in Windows using a virtual machine or Cygwin. This application is a set of GNU and open-source tools to provide functionalities of the Linux environment to the Windows operating system [57].For the sample project it was defined a TDD workflow in Figure 3.2. For every new test, developers had to change to Cygwin in order to run the unit tests, and then get back to Atmel Studio.

The workflow presented in Figure 3.2 does not encourage the use of TDD. This practice is hard to adapt and the minimal obstacle can be the reason for developers giving up TDD. I will present a seamless solution to run unit tests from Atmel Studio by integrating the Cygwin tools. The solution is explained in section 4.1.2 and intends to modify the software development workflow in Exatronic to another that allows TDD (see Figure 3.3).

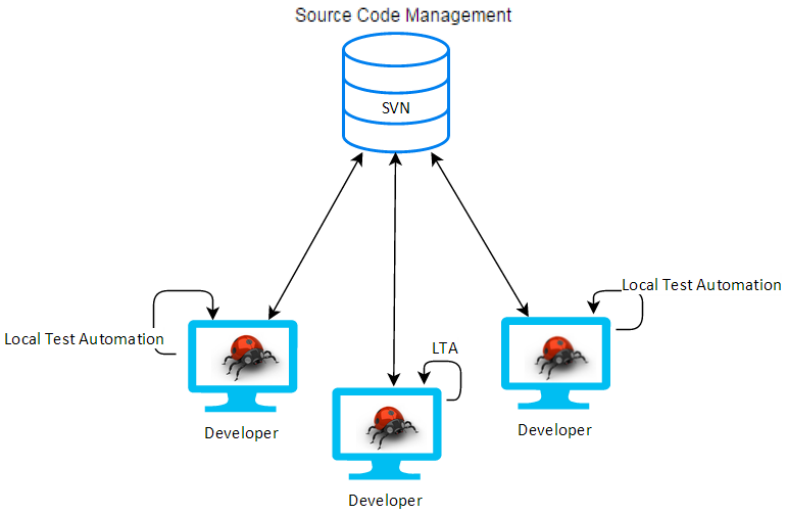


Figure 3.3: Workflow for the embedded software development with TDD

The above workflow concentrates the execution of unit testing only on the developers' machines. A test can pass in the local machine but if integrated with the other project modules it may fail. How can we know if the code committed to the repository wont broke the product code? The source code management system does not tell us if the code is broken. The solution is to implement a continuous integration server that will run all the unit tests, static analysis tool and will send feedback to all developers. Figure 3.4 shows the workflow with the CI server. This is the final workflow that includes TDD and CI and will be used as solution to have both practices on the embedded software development.

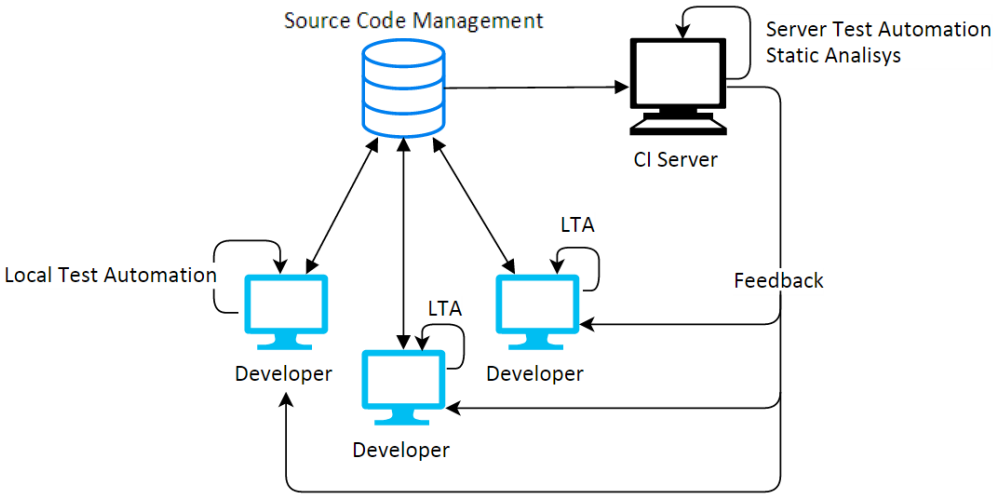


Figure 3.4: Workflow for the embedded software development with CI and TDD

In this solution, developers commit their code to the SCM system, the CI server will checkout the production code and run unit testing and static analysis tools on it. After that, the

CI server will send the build result to all developers. Jenkins was the choice to be the CI server, because it is open source, has high statistics of adoption, has a reliable development team that maintain the tool and there are multiple plugins to support building and testing virtually any project [58]. Jenkins is a server-based system written in Java that provides continuous integration services for software development [58].

The execution of static code analysis is planned to be made by using Cppcheck and Copy/Paste Detector (CPD) [59], [60]. They were chosen for being open-source, working with C language and for having plugins to work on Jenkins. There is a lack of open source and maintainable tools supporting C language. In this case, both tools are some of the few that fulfil these requirements. Cppcheck is a static analysis tool for C/C++, it searches for bugs that compilers normally fail to detect, for example, the use of null pointer dereferencing, incorrect use of functions, memory leaks, resource leaks, the use of obsolete functions, among others. The CPD tool finds duplicated code in one or different files for different programing languages. In order to complement the static analysis it will be used two more tools: a compiler warning detector and an open tasks scanner. They are both Jenkins plugins, and their execution occurs on the CI server. The compiler warning detector scans the console log or specified log files for warnings of different formats and reports which were found [61]. The open tasks scanner searches the workspace files for open tasks like TODO, FIXME, or @deprecated [62].

To summarize, the key improvement practices proposed in this dissertation are:

- Simplify the adoption of TDD, by using a setup that makes practical to run unit tests from the Atmel IDE;
- Introduce a continuous integration practice with additional code quality controls to run at the integration server;

The proposed strategy and setup will be tested using the project available from Exatronic and a project from James Greening book [63].

chapter 4

Agile Practices Implementation

In this chapter it will be will be described in detail the TDD and CI agile practices and the static analysis tools. It will be also recommended Git, a different version control system tool. For the TDD practice it will be described the CppUTest framework and how to use it in Atmel Studio. Furthermore, there will be a description of the CppUTest framework and how to use it in Atmel Studio. For the CI practice I will explain the server chosen and how to integrate the results from static tools and CppUTest with the server.

4.1 Test Driven Development

Test driven development is about writing a test before the code, forcing developers to think in the module before writing the code. After creating a test, the code is written to pass that test and then refactored to be clean and simple, but still passing the test. Bob Martin described over the years three simple rules that reinforces the idea of TDD [64]:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

The development code must start with something simple and then grow until the final product. However, the developer must resist writing new untested code because using the TDD practice is about making the development simple, writing only the necessary code to pass a test and not code that we may think that will be needed. It is an iteration process, a test at a time and some code to pass it.

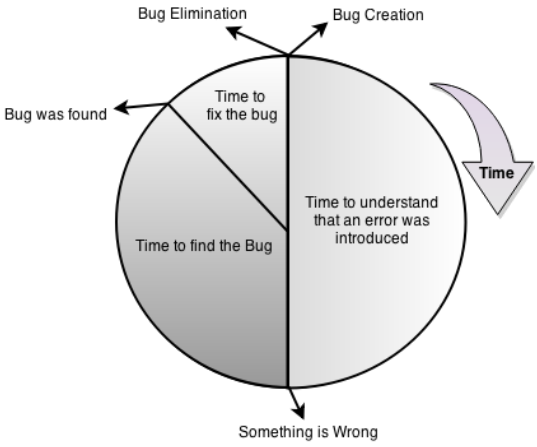


Figure 4.1: Bug lifecycle without TDD, adapted from [63]

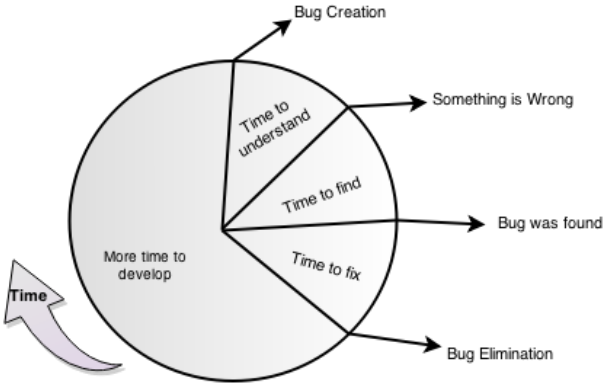


Figure 4.2: Bug lifecycle with TDD, adapted from [63]

Without TDD, finding a bug is more difficult, and if found, it normally happens later in the development. Analyzing Figure 4.1, when the time to understand a bug in our code is long, then the time to find it will increase. After discovering the bug, the difficulty to fix it will be higher due to the project complexity which increases over the time. On the other hand, in Figure 4.2, when the time to discover a bug in the code is short, then the time to find it will be smaller too and the effort to fix it will be low, saving more time for the development process.

To have these advantages in the development process we must follow five steps mentioned in the *Test-Driven Development for Embedded C* book [63]:

1. Add a small test.
2. Run all the tests and see the new one fail.
3. Make small changes needed to pass the test.
4. Run all test and see the new one pass.
5. Refactor to remove duplication and improve expressiveness.

Yet a problem in doing TDD for embedded systems is that we must interact with the hardware. It is possible to write tests off the target, it does not ensure that the code will work in

hardware but developers can understand and test how the code is supposed to behave in hardware. For more complex interactions with the hardware it is possible to fake some dependencies by using mocks, or stubs to take place of the production code and break the hardware dependencies which difficult the testing progress. A stub in programing language is a controllable replacement for an existing dependency in the system [63]. A mock is an object that has the same interface as a complex object that cannot be used in test [63]. The mock object simulates the behavior of the complex one, in order to be able to use it in tests. There are frameworks to create these mock objects, for example, CppUTest has one called CppUMock.

On the embedded TDD cycle presented in Figure 4.3, we are only on the stage one, where the unit tests are executed for every modification in the production code. The embedded TDD cycle introduces tests in the target hardware and not only in the development system. The stage two is about compiling to the target using the cross compiler for the production code. It prevents compiler incompatibilities and warns of porting problems such as unavailable header files, incompatible language support and missing language features [63]. The stage three is done only if the target hardware is not available. Running the unit tests in an evaluation board will reduce the risk because the compiled code can run differently in the development system and the target processor [63]. The same happens for the stage four, but here we have the target hardware to do the unit tests and it is possible to add more specific ones. The last stage is where the manual tests are created to test code that cannot be fully tested using automated tests [63].

Some stages may not be possible, stage four and five can only be done by having the target hardware earlier in the development and stage three can be skipped if there is already the target hardware. Anyway, the stage one is where most part of the code is written and tested, and it is the stage most frequently executed. For this reason I will only focus in the stage one of the embedded TDD cycle.

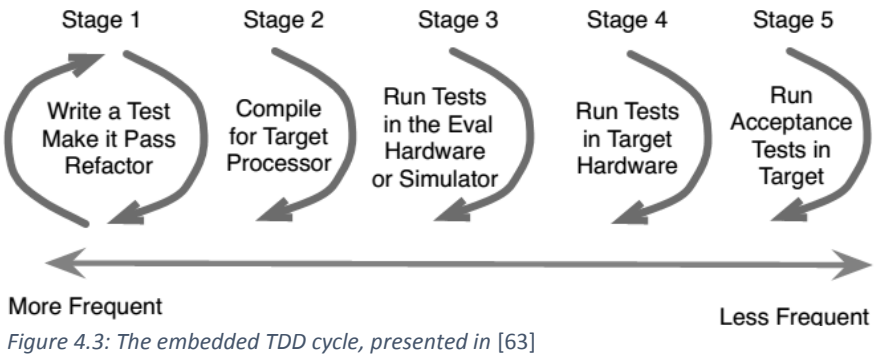


Figure 4.3: The embedded TDD cycle, presented in [63]

4.1.1 Unit Testing with CppUTest

CppUTest framework was developed with the purpose of being usable for embedded systems [65]. The framework uses a primitive subset of C++, making it a good choice for embedded software because not all compilers support the full language [65]. Although being a C++ framework, developers do not need to master the language. There are two ways to use CppUTest: downloading the source code or installing through the repository manager.

If using the repository manager, the framework will be installed automatically. If using CppUTest source code, it is necessary to install it manually. The installation and execution can be done using Cygwin command-line or the Linux shell. To install it, it is necessary to execute the configure file using the command ***./configure*** inside the source folder. The next step is simply to build the framework using the ***make*** command. CppUTest has integrated tests to verify if the framework was successfully installed. They can be ran by executing the ***make test*** command after the installation process. If all tests pass, then the framework is ready to be used. For beginners with CppUTest, there is a useful set of scripts that automatically creates the structure of C projects with examples of unit tests and makefiles. In the Annex C there is a description on how to install and execute these scripts.

To have the TDD practice in the development process it is necessary to create an environment to allow it. The development project folder must have inside two folders, one for production code and another for test code. The test folder must have, at least, two files. CppUTest needs one file to write tests and another, the main, to execute them automatically. The main structure of a test file can be seen in Figure 4.4.

The test execution order is irrelevant because each test is independent, they do not need the execution of other tests to have the expected result. If a group of tests needs to run some code before or after the execution there is a place for it, the ***TEST_GROUP()*** function. Each test belongs to a group, previously defined as ***TEST_GROUP(GroupName)*** and inside of the group definition, there are two methods to execute code before, **setup(),** and after, **teardown()**. With this, it is possible to have, in the same file, different tests with different dependencies. It is also possible to ignore some tests just by using the macro ***IGNORE_TEST()*** instead of ***TEST()***. The ignored tests will be compiled but not executed and will appear in the result as ignored.

```
extern "C"
{
// #includes for things with C linkage
}

// #includes for things with C++ linkage


#include "CppUTest/TestHarness.h"

TEST_GROUP(GroupName)
{
    //Define data accessible to test group members here.

    void setup()
    {
        //initialization steps are executed before each TEST
    }
    void teardown()
     {
         //clean up steps are executed after each TEST
     }
};

//Many test cases like this can be defined in the test file.
TEST(GroupName, TestCaseName)
{
    /*
    * The test case contains:
    *     test specific initializations
    *     operations on the code under test
    *     test specific condition checks
    */
}
```

Figure 4.4: CppUTest Test File structure presented in [63]

The C files that cannot compile using the C++ compiler can be included by placing them inside the ***extern*** C class, as presented in Figure 4.4.

As mentioned before, to execute all tests it is necessary to have a test main and there is no other step to run the execution of all tests, there is no installation process or something alike. Normally the main is never modified and will always look like Figure 4.5.

```
#include "CppUTest/CommandLineTestRunner.h"

int main(int argc, char** argv)
{
    return RUN_ALL_TESTS(argc, argv);
}
```

Figure 4.5: Main to run all CppUTest Tests presented in [63]

As other xUnit tools, each test must have conditional checks to verify if the operations in the code under test have the expected result. CppUTest provides many of them that can be found in the Annex C.

Finally, to execute all the tests it is necessary to compile them using a makefile. There are two ways to compile tests using CppUTest: importing the CppUTest makefile or creating a makefile from scratch. The easy way is importing the CppUTest makefile called ***MakeFileworker.mk***[1], because the only thing we must do is fulfil the available macros. These macros can be found in the CppUTest makefile along with the description of each one (see Annex C). The Figure 4.6 shows a skeleton of the makefile that uses the ***import*** command. Apart from the paths to the source and test code, it is required to add the path to the folder where CppUTest is installed (***CPPUTEST_HOME***) and one of the included directories must be the CppUTest include folder.

```
1
2   COMPONENT_NAME =
3   CPPUTEST_HOME = Path/to/cpputest
4
5   CPPUTEST_USE_EXTENSIONS = Y
6
7   CPP_PLATFORM = gcc
8
9   SRC_DIRS =
10
11  TEST_SRC_DIRS = \
12
13  INCLUDE_DIRS =\
14      $(CPPUTEST_HOME)/include \
15
16  CPPUTEST_WARNINGFLAGS =
17
18  CPPUTEST_CFLAGS =
19  CPPUTEST_CXXFLAGS =
20  LD_LIBRARIES =
21
22  include $(CPPUTEST_HOME)/build/MakefileWorker.mk
```

Figure 4.6: Makefile skeleton to compile CppUTest tests (with import)

On the other hand, is possible to not use the import of ***MakefileWorker.mk***, being a more complex alternative because it is necessary to write the rules to compile it. The required definitions made in Figure 4.6 must be added too, however it is necessary to add the location of the CppUTest libs, as shown highlighted in Figure 4.7.

[1] It can be found in the cpputest/build directory.

```
1   INCLUDE_DIRS =\
2       $(CPPUTEST_HOME)/include \
3
4   SRC_DIRS = \
5
6   CPPUTEST_HOME = path/to/CppUTest
7   COMPONENT_NAME =
8
9   CFLAGS =
10  CXXFLAGS =
11  CPPFLAGS =
12
13  LDLIBS = -lCppUTest
14  LDFLAGS = -L$(CPPUTEST_HOME)/lib
15
16  all:  $(COMPONENT_NAME) run
17
18  ### Rules to compile source and test files ###
19
20  run: $(COMPONENT_NAME)
21      ./$(COMPONENT_NAME)
22
23  clean:
24      $(RM) -R $(COMPONENT_NAME) $(OBJ_DIR)
```

Figure 4.7: Makefile skeleton to compile CppUTest files (without import)

Anyway, if CppUTest is installed using the *repository manager*, it will not be possible to import the makefile from CppUTest, because the **MakefileWorker.mk** will not exist. The only way is support the installation with the source code or create the makefile from scratch. However, for this makefile, compared to the one in Figure 4.7, it is only necessary to add the path to the CppUTest lib[2] as highlighted in Figure 4.8, because the CppUTest include files, after the installation, are part of the system.

```
2   INCLUDE_DIRS =\
3       $(CPPUTEST_HOME)/include \
4   SRC_DIRS = \
5
6   CPPUTEST_LIB = path/to/CppUTest/libCppUTest.a
7   COMPONENT_NAME =
8
9   CFLAGS =
10  CXXFLAGS =
11  CPPFLAGS =
12
13  LDLIBS = -lCppUTest
14  LDFLAGS = -L$(CPPUTEST_LIB)
15
16  all:  $(COMPONENT_NAME) run
17
18  ### Rules to compile source and test files ###
19
20  run: $(COMPONENT_NAME)
21      ./$(COMPONENT_NAME)
22
23  clean:
24      $(RM) -R $(COMPONENT_NAME) $(OBJ_DIR)
```

*Figure 4.8: Makefile skeleton to compile CppUTest files if the framework was installed using the **apt-get** command*

[2] The lib location for Ubuntu 14.04 32bits is: usr/lib/i386-linux-gnu/libCppUTest.a

Now, in order to get the test results, this type of makefiles needs to have a rule to run the generated CppUTest output file manually. Figure 4.7 and Figure 4.8 are examples, the target ***run*** is a dependency of the main target ***all,*** being always executed after the compilation. For the other type of makefiles, the output file is executed automatically. In addition, to use the CppUTest results in a CI server it is necessary to have them in a ***XML*** file. CppUTest provides the possibility to create a ***XML*** file just by using the argument ***–o junit*** when executing the CppUTest output file. For example, in Figure 4.8 it would be enough modify the content of the ***run*** target to ***./$(COMPONENT_NAME) –o junit***.

All the information about how to create the makefile to compile the CppUTest files is summarized in the Table 4.1. The table title **import?** asks if the makefile will be created importing the ***MakefileWorker.mk***, the title ***apt-get?*** asks if CppUTest was installed using apt-get. Then, there are the main additions to make for each case, as well as the modifications in order to get the results on shell or in a ***XML*** file.

Import?	**apt-get?**	**Main Additions**	**Results**	
			In Shell	**XML File**
Yes	No	**1.** Add the source directory location in ***CPPUTEST_HOME*** **2.** Add ***$(CPPUTEST_HOME)/include*** to the include dirs. **3.** Include the makefile: ***$(CPPUTEST_HOME)/build/MakefileWorker.mk***	**Automatic**	Create it manually. **Ex:** ./App_Tests –o junit
No	No	**1.** Add the source directory location in ***CPPUTEST_HOME*** **2.** Add ***$(CPPUTEST_HOME)/include*** to the include dirs. **3.** Add the flag: **-L*$(CPPUTEST_HOME)/lib*** **4.** Add the lib: **-lCppUTest.**	**1.** Create a target to execute the output file. **2.** Place the target as a main dependency.	**1.** Create a target to execute the output file with the xml arguments. **2.** Place the target as a main dependency.
No	Yes	**1.** Add the flag to the lib location of **libCppUTest.a.** **Ex: -L$(CPPUTEST_LIB),** with the variable defined with the path.	**1.** Create a target to execute the output file. **2.** Place the target as a main dependency.	**1.** Create a target to execute the output file with the xml arguments. **2.** Place the target as a main dependency.

Table 4.1: Guide to know the essential additions to place in the makefile that compiles CppUTest files

After explaining how to install CppUTest, create tests and compile them, I will use the framework in a simple example: the factorial calculation of a number. Before writing code, it is necessary to make a list of things that we want to test. The best way to organize it is by making a list:

- The input value cannot be a negative integer;
- The factorial of a number is the product of all the whole numbers from that number to one, for example, 3! = 3x2x1 = 6;
- The factorial of zero is one (0!=1);

With this I created three test cases as shown in Figure 4.9. The full implementation can be seen in Annex C.

```
15
16  TEST(factorial inputNegative)
17  {
18      int input = -1;
19      long result = factorial(input);
20      LONGS_EQUAL(0, result);
21  }
22
23  TEST(factorial, testRandomFactorial)
24  {
25      int input = 3;
26      long result = factorial(input);
27      LONGS_EQUAL(6, result);
28  }
29
30  TEST(factorial, factorialOfZero)
31  {
32      int input = 0;
33      long result = factorial(input);
34      LONGS_EQUAL(1, result);
35  }
36
```

Figure 4.9: Factorial test cases

The next step is the compilation process. The ***MakefileWorker.mk*** was imported into my makefile as shown in Annex C. The compilation result is presented in Figure 4.10. There are failed tests and one who happens to pass. It is suppose to have failed tests because there is no code written yet, only an empty function. As seen in Figure 4.10, CppUTest result informs which tests failed, the expected value and the value obtained, as well as the number of all tests, which ones ran, which ones were ignored and other information. The label ***checks*** count the number of conditional checks (assertions) executed, such as ***LONGS_EQUAL()***, and the label ***filtered out***, as the name says, shows the number of tests filtered out using command-line options.

Now, I must write only the necessary code to pass the tests (see Annex C for the source code). The final build result can be seen in Figure 4.11, the label ***Errors*** in the Figure 4.10 was replaced by the label **OK**, it means that all tests passed.

```
tfa@tfa-VirtualBox:~/Documents/svn/first_project/Tests/Nova pasta$ make

objs/src/factorial.o objs/tests/AllTests.o objs/tests/factorial_Tests.o
Linking Factorial
g++ objs/src/factorial.o objs/tests/AllTests.o objs/tests/factorial_Tests.o    -
l/usr/lib/i386-linux-gnu/libCppUTest.a -o Factorial -lCppUTest -lCppUTestExt -lp
thread -lstdc++
./Factorial

tests/factorial_Tests.cpp:39: error: Failure in TEST(factorial, factorialOfZero)
        expected <1 0x1>
        but was  <0 0x0>

.
tests/factorial_Tests.cpp:32: error: Failure in TEST(factorial, testRandomFactor
ial)
        expected <6 0x6>
        but was  <0 0x0>

..
Errors (2 failures, 3 tests, 3 ran, 3 checks, 0 ignored, 0 filtered out, 1 ms)

make: *** [run] Error 2
tfa@tfa-VirtualBox:~/Documents/svn/first_project/Tests/Nova pasta$
```

Figure 4.10: First build result

```
Tiago@Tiago-PC /cygdrive/c/Users/Tiago/Dropbox/Tese/Escrita_Tese/Anexos/Nova pas
ta
$ make
compiling factorial_Tests.cpp
mkdir -p objs/tests/
g++  -g    -include C:\Users\Tiago\Desktop\Local_Test\cpputest/include/CppUTest/
MemoryLeakDetectorNewMacros.h  -DCPPUTEST_COMPILATION   -include C:\Users\Tiago\
Desktop\Local_Test\cpputest/include/CppUTest/MemoryLeakDetectorMallocMacros.h  -
Iincl  -IC:\Users\Tiago\Desktop\Local_Test\cpputest/include    -c -MMD -MP -o ob
js/tests/factorial_Tests.o tests/factorial_Tests.cpp
compiling AllTests.cpp
mkdir -p objs/tests/
g++  -g    -include C:\Users\Tiago\Desktop\Local_Test\cpputest/include/CppUTest/
MemoryLeakDetectorNewMacros.h  -DCPPUTEST_COMPILATION   -include C:\Users\Tiago\
Desktop\Local_Test\cpputest/include/CppUTest/MemoryLeakDetectorMallocMacros.h  -
Iincl  -IC:\Users\Tiago\Desktop\Local_Test\cpputest/include    -c -MMD -MP -o ob
js/tests/AllTests.o tests/AllTests.cpp
compiling factorial.c
mkdir -p objs/src/
cc  -g   -DCPPUTEST_COMPILATION   -include C:\Users\Tiago\Desktop\Local_Test\cpp
utest/include/CppUTest/MemoryLeakDetectorMallocMacros.h  -Iincl  -IC:\Users\Tiag
o\Desktop\Local_Test\cpputest/include    -c -MMD -MP  -o objs/src/factorial.o sr
c/factorial.c
Building archive lib/libFactorial_Test.a
mkdir -p lib/
ar rvc lib/libFactorial_Test.a objs/src/factorial.o
a - objs/src/factorial.o
ranlib lib/libFactorial_Test.a
Linking Factorial_Test_tests
g++ -o Factorial_Test_tests objs/tests/factorial_Tests.o objs/tests/AllTests.o l
ib/libFactorial_Test.a C:\Users\Tiago\Desktop\Local_Test\cpputest/lib/libCppUTes
t.a  -static -g
START_TIME=1432780826
rm -f objs/src/factorial.gcda objs/tests/factorial_Tests.gcda objs/tests/AllTest
s.gcda gcov_output.txt gcov_report.txt gcov_error.txt ; echo "Running Factorial_
Test_tests"; ./Factorial_Test_tests
Running Factorial_Test_tests
...
OK (3 tests, 3 ran, 3 checks, 0 ignored, 0 filtered out, 1 ms)


Tiago@Tiago-PC /cygdrive/c/Users/Tiago/Dropbox/Tese/Escrita_Tese/Anexos/Nova pas
ta
$
```

Figure 4.11: Factorial test results

4.1.2 Unit Testing and Atmel Studio

Atmel Studio is based on GCC, the GNU C compiler, as many of the current microcontroller programming environments[54]. It provides a large collection of embedded software for Atmel microcontrollers named Atmel Software Framework (ASF)[54]. To make possible testing embedded projects using Atmel Studio it is necessary to make some modifications, first in the header files from the ASF, second in the Atmel Studio.

a. Adapting ASF header files

The code from the Atmel Software Framework has definitions of memory and register locations named Special Function Registers (SFR). The Atmel Studio allows addressing these registers, for example, IO registers, treating them as data memory, being useful to use it in higher level languages.

```
#define PORTC    (*(PORT_t *) 0x0640)
```

Figure 4.12: Example of an IO register definition

The above figure shows an example of pointing the register PORTC to the address 0x0640. The register points to a physical address from the target hardware and will throw memory access errors during the execution of unit tests. To execute unit tests in software to embedded systems it is necessary to isolate the code under test from the target hardware. To isolate the access to the SFR memory mapping using the ASF it is necessary to adapt some files without harming the reliability of tests. These files are: the main header file, ***io.h,*** where are included a set of headers files that uses the SFR memory mapping; the ***sfr_defs.h*** which defines macros for accessing the special function registers; and the ***ioxxxx.h,*** responsible of the definitions for the respective microcontroller. The last two headers are included by ***io.h*****.**

<avr/io.h>

This header file includes the appropriate IO definitions for the device that has been specified. To do unit testing, the include of ***portpins***, ***common***, ***version***, ***fuse*** and ***lock*** header files must be blocked [66]. It is possible to block using a macro condition as represented in Figure 4.13.

```
#ifndef Unit_Testing
      #include <avr/portpins.h>
      #include <avr/common.h>
      #include <avr/version.h>
      #include <avr/fuse.h>
      #include <avr/lock.h>
#endif
```

Figure 4.13: <avr/io> include files to block

It is recommended to use a macro name such as ***UNIT_TESTING*** or the name of the framework, to be self-explanatory and obvious. It is also possible to do it just by commenting the unwanted code lines.

<avr/sfr_defs.h>

This file is included by all the ***ioxxx*** header files. On the inside there are macros to make the special function register definitions look like C variables. Some of these definitions, presented in Figure 4.14, need to be adapted. They are the ***_MMIO_xxxx*** variables, because they perform direct access to hardware memory being used, indirectly, in the respective ***ioxxxx***, the header of a specific microcontroller [66].

```
//Original Code
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _MMIO_WORD(mem_addr) (*(volatile uint16_t *)(mem_addr))
#define _MMIO_DWORD(mem_addr) (*(volatile uint32_t *)(mem_addr))
//After Adaptation
#define _MMIO_BYTE(mem_addr)  (uCmemory[mem_addr])
#define _MMIO_WORD(mem_addr)  (*(volatile uint16_t *) &uCmemory[mem_addr])
#define _MMIO_DWORD(mem_addr) (*(volatile uint32_t *) &uCmemory[mem_addr])
```

Figure 4.14: Direct access to memory, the original code and the adapted for tests.

The ***_MMIO_xxxx*** registers need to use a fake memory which can be done just by creating a simple array. The code adapted in Figure 4.14 mentions an array named ***uCmemory*** that fakes the microcontroller memory, as shown in Figure 4.15. Here it is being used a stub to simulate the microcontroller data memory. In this case, using a stub allows testing code without dealing with the dependency directly. The array definition can be a new test header file, named ***uCmemory*** for example, or just can be defined in the ***sfr_defs*** header file.

```
#define MEM_SIZE 128000
#uint8 t uCmemory[MEM SIZE];
```

Figure 4.15: A stub to fake a microcontroller memory

<avr/ioxxxx.h>

For simple processors, like ***megaAVR*** 8-bit series, there is no need to adapt **ioxxxx**, because, contrary to the more complex ones, for example, ***AVR XMEGA*** 8-bit series, the registers are created using flat structures as represented in Figure 4.16 [66].

```
/* Port C */
#define PINC    _SFR_IO8(0x13)
#define DDRC    _SFR_IO8(0x14)
#define PORTC    SFR IO8(0x15)
```

Figure 4.16: Structure of a register in a simple processor

For more complex microcontrollers, **ioxxxx** have C structures that point to specific locations in memory [66]. In such cases, the header file needs to be adapted. For example, the adaptation of the register from Figure 4.12, would be such as in Figure 4.17.

```
#define PORTC    (*(PORT_t *) &uCmemory[0x0640])  /* Port C */
```

Figure 4.17: Register adapted to allow unit testing

All these modifications can be done by commenting on the original code and adding the adapted one instead, or by creating a macro condition that uses the original code if the macro is not an argument while compiling. This is useful if we want to use the same header files to do tests and to develop code.

These modification on the three mentioned header files will allow testing Atmel embedded projects. With all files adapted there is no wrong memory access while using the registers.

b. Setting Atmel studio up for using CppUTest framework

This section describes how to adapt Atmel studio to be able to run unit tests using CppUTest, step by step.

Step 1. The first step is to create an Atmel solution project. Inside the solution two projects must be created: one for the production code and another to the unit tests (see Figure 4.18 as example). The template and device choice for the project that will run tests can be the same as the other project, but ends up being irrelevant because it will only run tests.

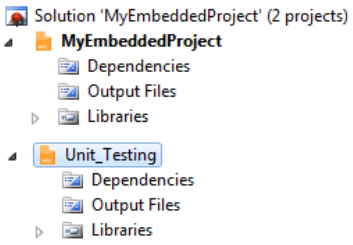


Figure 4.18: Solution with two projects

Step 2. The project for the embedded code does not need any modifications, it must be developed as it was before using CppUTest on Atmel Studio. I will refer the project that will contain tests as the unit testing project. In this project it is necessary to add the files adapted in section a, but as seen in that section, they are imported using the standard include directive in ASF named ***<avr>.*** I need to simulate the existence of this folder in the unit testing project in order to include the adapted files and not the original ones.

So, in this step, it is necessary to create a folder which can be called ***avr_include*** for example, and on the inside a new folder called ***avr***. The adapted files mentioned in section ***a*** must be added here. It must be added not only the adapted files, but also all the ASF header files needed to compile the code under test. Normally theses files are only three, ***io***, ***sfr_defs*** and ***ioxxxx***. It is really important to add only the needed files because of three issues, the files from the CppUTest

framework and ASF have some identical names for macros, the ASF has identical files to the default GCC include system files used by CppUTest framework and also because the path to the ASF files has blank spaces that CppUTest framework cannot handle. For example, Figure 4.19 shows the unit testing project for an ***ATmega32*** device and although the adapted header files are only ***io*** and ***sfr_defs*** it is also necessary to include ***iom32*** to compile the code under test. In addition, if the stub to fake the microcontroller was placed in a header file, it must be added to the folder ***avr_include.***

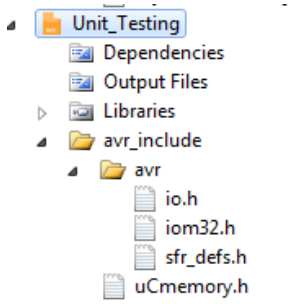


*Figure 4.19: The inside of the **avr_include** folder*

Step 3. In this step it must be created the file where the tests will be written (see Figure 4.4) and the main file to run them (see Figure 4.5). After this step, the Atmel solution structure needed to execute tests is complete. Figure 4.20 show an example of a complete structure for a unit testing project.

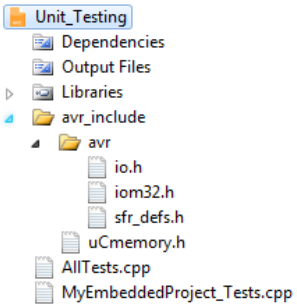


Figure 4.20: An example of a complete project structure to develop tests, using CppUTest, in Atmel studio.

Step 4. This step is about the compilation process for the unit testing project. The first approach was creating a static library with the CppUTest framework inside the Atmel solution. However, it did not work because Atmel studio has a cross-compiler for CPU and operating system, Windows, and it is limited, without the console I/O or system calls, for example. Also, as mention before, the files from the CppUTest framework and ASF have some identical names for macros and because of this, the compilation will throw more errors. This was an obstacle because CppUTest

assumes a POSIX environment to work, Atmel studio, as it is, cannot compile CppUTest files. The second approach was to compile it using an external makefile, and it worked. But first, for Atmel studio to run the external makefile successfully, it needs a different environment to work, such as provided by Cygwin.

In this step it is necessary to install Cygwin on the machine and then add the Cygwin bin folder to the Windows environment variable[3] ***Path***. Atmel studio shell is based on the Windows shell, adding the Cygwin path to the Windows environment makes Cygwin tools available for the Atmel studio shell. Now, Atmel studio is able to compile the unit testing project.

Step 5. This step is dedicated to the external makefile. To make Atmel studio compile a project using an external makefile it is necessary to go to the project configurations and on the ***build*** separator, check the option to use an external makefile. Then, it must be created a makefile for the unit testing project. It can be used one of the two types of makefiles mentioned in section 4.1.1. If the makefile includes the CppUTest makefile it is necessary to make some modifications:

First. When cleaning the unit testing project, the makefile will delete, if exists, a file named ***cpputest_*.xml***, but the Atmel Studio shell, even integrated with Cygwin, does not recognize the wildcard and will throw an error. The solution is to delete this argument from the cleaning command in the included CppUTest makefile (see Figure 4.21). This action will not harm the reliability of tests because this file will not be used for now.

```
413  DEP_FILES = $(call src_to_d, $(ALL_SRC))
414  STUFF_TO_CLEAN += $(DEP_FILES) $(PRODUCTION_CODE_START) $(PRODUCTION_CODE_END)
415  STUFF_TO_CLEAN += $(STDLIB_CODE_START) $(MAP_FILE) cpputest_*.xml junit_run_output
416
```

*Figure 4.21: Line code from **MakefileWorker.mk** where the argument **cpputest_*.xml** needs to be deleted.*

Second. The second modification is in the Cygwin bin folder. During the unit project compilation, an error will be thrown when compiling C files. This happens because CppUTest makefile uses the command ***cc*** to compile C files, and in the Cygwin bin folder, ***cc*** is a Cygwin symbolic link to ***gcc.exe***. This is a problem because Atmel studio is not able to read Cygwin-created symbolic links. The solution is to replace that symbolic link for a Windows symbolic link to the Cygwin ***gcc.exe***.

[3] The windows environment variables can be found in Control Panel/ System, on Windows 7.

After these two modifications it is possible to compile the unit testing project using the makefile that imports another from CppUTest. Figure 4.22 is an example of a makefile structure to compile the unit testing project mentioned in Figure 4.20

```
######### MAKEFILE USING IMPORT #######

COMPONENT_NAME = MyEmbeddedProject
CPPUTEST_HOME = C:\Users\Tiago\Desktop\Local_Test\cpputest

AVRDEVICE=AVR_ATmega32

SRC_DIRS = \
MyEmbeddedProject \


TEST_SRC_DIRS = \
Unit_Testing \

INCLUDE_DIRS =\
$(CPPUTEST_HOME)/include  \
Unit_Testing/avr_include \

CPPUTEST_WARNINGFLAGS = -Wall
CPPUTEST_WARNINGFLAGS +=

CPPUTEST_CFLAGS =
CPPUTEST_CFLAGS +=

CPPUTEST_CXXFLAGS += -D__$(AVRDEVICE)__
CPPUTEST_CXXFLAGS += -DCPPUTEST

include $(CPPUTEST_HOME)/build/MakefileWorker.mk
```

Figure 4.22: Example of a makefile using import

On the other hand, if the makefile was made from scratch, it must be ensured that the tests are executed after the compilation and no more modifications are needed. For example, Figure 4.23 executes the CppUTest output file generated after the compilation.

```
80
81  run: $(COMPONENT_NAME)
82      ./$(COMPONENT_NAME)
83
```

Figure 4.23: Commands to run the CppUtest output file

Regardless of the makefile type, the recommended directory to place it is in the same path as the solution project, because it makes easier the use of both project paths, as seen in Figure 4.22.

To summarize, in this step it is necessary to check the Atmel studio option to use an external makefile for the unit testing project, create a makefile, make some modifications if necessary and place it on the solution project directory. After this, in the unit testing project proprieties browse to the location of the makefile and save it (see Figure 4.24).

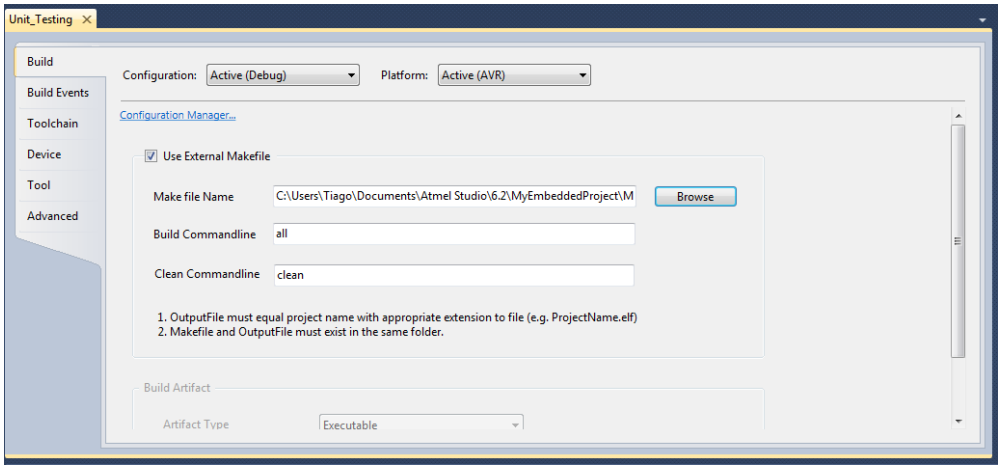


Figure 4.24: Unit testing project properties

Step 6. Now, Atmel studio is ready to compile the solution project and get the test results for each build. After compiling, the test results will always be executed and printed in the Atmel studio output window, and if one or more tests fail, it will appear in the error list window as a normal error. A failed test will fail the build and the expected result from the test will be presented in the output window. Figure 4.25 and Figure 4.26 are an example of how Atmel studio will show the results for the example used through these six steps.

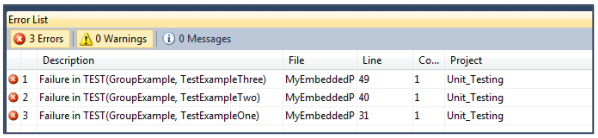


Figure 4.25: Error list containing the failed tests

```
        START_TIME=1433960001
        rm -f objs/Unit_Testing/AllTests.gcda objs/Unit_Testing/MyEmbeddedProject_Tests.gcda gcov_output.txt gcov_report.txt gcov_error.txt ; echo "Running MyEmbe
MyEmbeddedProject_tests
        Running MyEmbeddedProject_tests
        .
C:\Users\Tiago\Documents\Atmel Studio\6.2\MyEmbeddedProject\Unit_Testing\MyEmbeddedProject_Tests.cpp(49,1): error: Failure in TEST(GroupExample, TestExampleThree)
            expected <hi>
            but was  <bye>
            difference starts at position 0 at: <          bye       >
                                                 ^
        .
C:\Users\Tiago\Documents\Atmel Studio\6.2\MyEmbeddedProject\Unit_Testing\MyEmbeddedProject_Tests.cpp(40,1): error: Failure in TEST(GroupExample, TestExampleTwo)
            expected <1 0x1>
            but was  <0 0x0>
        .
C:\Users\Tiago\Documents\Atmel Studio\6.2\MyEmbeddedProject\Unit_Testing\MyEmbeddedProject_Tests.cpp(31,1): error: Failure in TEST(GroupExample, TestExampleOne)
            expected <  0 0x00>
            but was  <255 0xff>
        .
        Errors (3 failures, 4 tests, 4 ran, 4 checks, 0 ignored, 0 filtered out, 117 ms)
        make: *** [all] Error 3
        make: Leaving directory `C:/Users/Tiago/Documents/Atmel Studio/6.2/MyEmbeddedProject'
    Done executing task "RunCompilerTask" -- FAILED.
Done building target "CoreBuild" in project "Unit_Testing.cproj" -- FAILED.
```

Figure 4.26: Output window with the CppUTest execution result

4.2 Source Code Management

A source code management system is a repository of files that records changes over time to be available the recuperation of a specific version later. The repository is where the source code of products under development will be. Every change made to the source is tracked, along with who made the change, a message to explain why, and references to fixed problems, or enhancements. The source code management systems can be divided into two categories: centralized and distributed. Centralized systems have a single central copy of the code on a server and developers commit changes to this central copy only. In distributed systems, every developer can have a copy of the code, with full history of the project.

I will focus on two source code management tools, SVN and Git. SVN because, as mention before, Exatronic uses it. GIT because is the new SCM emerging and it was initially developed by the creator of Linux kernel, Linus Torvalds [67]. Both are free and open source. SVN and Git are different, SVN has a centralized repository and Git has a distributed source control.

Git is the SCM tool recommended for the benefits that it adds comparing to SVN. For example, it adds the possibility to use different development workflows [67]. Git does not depend on a centralized server, but does have the ability to synchronize with other Git repositories, for example, it is possible to have multiple repositories to the same project to push (commit) and pull (update) changes between them. Git also gives to every developer their own local copy of the entire project [67]. Having a local copy creates an isolated environment that allows every developer to work independently of all the other changes in the main repository. Additionally, Git has a new efficient and faster branching and merging model comparing to SVN [67].

In order to stay with actual workflow in Exatronic, it can be used the centralized workflow which is equal to the SVN workflow, but with the mentioned Git advantages. However, the better option to use is a mix between this workflow and the feature branch workflow. A branch is an independent line of development, as represented in Figure 4.27. For example, new branches are created to add a new feature or fix a bug, then the changes are made only on the respective branch, ensuring that unstable code is never committed to the main code, giving an opportunity to clean up before merging it to the main branch. Figure 4.27 presents a repository with two isolated lines of development. Having them in branches makes possible to work on both in parallel, keeping the main code branch safe from questionable code. Git branches have an implementation much more lightweight than SVN, instead of copying files from directory to directory, Git stores a branch as a reference to a commit [68].

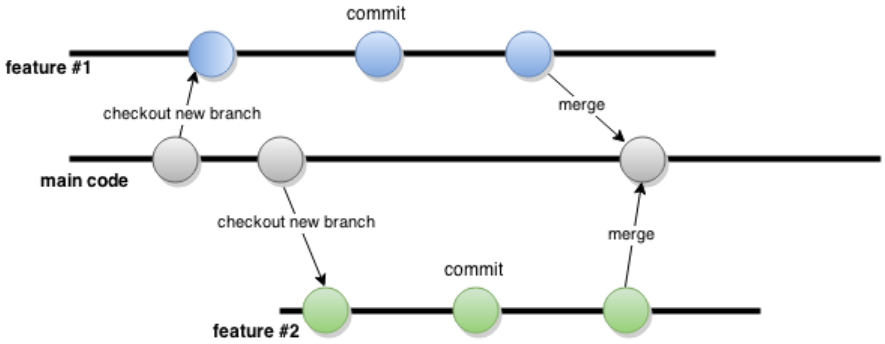


Figure 4.27: Workflow example of a project using Git

With Git is possible to send only the developed branch to the remote repository, then the CI server will check it for errors before merging with the main code. This will be described later in section 4.4. There is a Git application for Windows with a command-line interface where is possible to use all the Git commands as it was on Linux. Another alternative is using Cygwin with the development pack installed.

As seen in this section, a SCM tool makes the code visible for all development team giving the a partial idea of collective ownership.

4.3 Static analysis tools

In a general understanding, static code analysis is the practice of analyzing source code without running it. The GNU compiler collection (GCC) is an example, because it can detect lexical, syntactic and some semantic errors [69]. The tools used to do static analysis will complement the compiler, they will be like an extension of the compiler technology, and will indicate the need for refactoring. In this section it will be only mentioned Cppcheck and the Copy/Paste Detector (CPD), because the others are Jenkins plugins and do not have a specific application to run them. These tools will be configured to run in a shell, regardless the operating system. For Windows, they can run using Cygwin, in Linux, using the integrated shell.

a. Cppcheck

The current version of Cppcheck is 1.65. If using a Windows machine, it is not required to install it because Cppcheck is included on the Cygwin development package. For other machines, there is a Debian and Ubuntu package available for Cppcheck and it can be simply installed by using the ***apt-get*** command.

Cppcheck not only detect syntax errors like the C compiler, but also the type of bugs that the compilers normally fail to detect [60]. For example, the use of null pointer dereferencing, incorrect use of functions, memory leaks, resource leaks, use of obsolete functions, among others [60]. The resulting issues given by this tool are categorized in six types of severities represented in Table 4.2

SEVERITY	MEANING
error	Bugs in the code.
warning	Suggestions about defensive programming to prevent bugs.
style	Stylistic issues related to code cleanup[4].
performance	Suggestions for making the code faster[5].
portability	Portability warning[6].
information	Informational messages about checking problems.

Table 4.2: Different types of Cppcheck issues.

An example of a command-line to execute the Cppcheck is represented in Figure 4.28. This command includes the option to write the analysis result into a ***XML*** file, to be used by a CI server.

```
cppcheck --enable=all --inline-suppr --suppress=<issueid> --inconclusive
--xml-version=2 -I /path/to/dir/with/headers /path/to/dir/with/source/code 2>
path/to/save/report/filename.xml
```

Figure 4.28: Command line to execute Cppcheck

[4] Unused functions, redundant code, constness…
[5] These suggestions are only based on common knowledge.
[6] For example, code might work different in other compilers.

OPTION	MEANING
`--enable=all`	All the issues types are searched in the analyses.
`--inline-suppr`	Enable inline suppressions.
`--suppress=<issueid>`	Suppress all type of issues found by Cppcheck. <issueID> is the issue identity.[7]
`--inconclusive`	Allow that Cppcheck reports even though the analysis is inconclusive.
`--xml-version=2`	Writes results in xml version 2 format.
`-i /path/to/dir/with/headers`	Path to search for include files.
`/path/to/dir/with/source/code`	Path to search for code.
`2> path/to/save/report/filename.xml`	Normal shell redirection for piping output to a file.

Table 4.3: Options used to execute Cppcheck

Based on the information obtained from Cppcheck manual [60], I created the Table 4.3 to explain the options used to execute the tool. As shown in the table, it is possible to suppress specific issues from being shown in the results. There are some significant issues where is useful their suppressing. For example, when using standard C headers, Cppcheck cannot find them and even if their path is known, it is a waste of time to analyze them, because they are not our code and, in general, they are free of errors. The issue identity to suppress this is ***missingIncludeSystem***.

Another way to suppress issues, now specifically in the source code, is to add a comment before the code line that will throw an issue. This type of suppression are named inline suppressions and must be activated in the command-line, as it was in Figure 4.28. The way to use it is presented in following figure.

```
char a[10];
//cppcheck-suppress arrayIndexOutOfBounds
a[10] = 0; /* code issue */
```

Figure 4.29: Inline suppression using Cppcheck

[7] The issue identity can be found on the xml version 2 output, or using the option --template "{file}({line}): {severity} ({id}): {message}" on the command-line.

In order to test Cppcheck, it was made a simple example that can be seen on the Annex B. The example was made to throw the following issues:

- Function ***destroyA*** frees the same memory twice.
- In function ***a***, an array index is accessed out of bounds two times.
- In function ***z***, the variable ***i*** can be assigned inside the ***if*** condition.
- Functions and variables not used.
- Variables not initialized.

To have a better representation of what Cppcheck is capable, the result analysis of this tool will be compared to the compilation result using GCC 4.8.3 version. The results of Cppcheck are shown in Figure 4.31, while the results of the compiler are presented in Figure 4.30.

```
Tiago@Tiago-PC ~/sampleC/test
$ make
compiling AllTests.cpp
compiling CheckErrorsTest.cpp
compiling FactorialTest.cpp
compiling CheckErrors.c
src/cppCheck/CheckErrors.c: In function ‘z’:
src/cppCheck/CheckErrors.c:22:7: warning: variable ‘i’ set but not used [-Wunused-but-s
et-variable]
   int i;
       ^
src/cppCheck/CheckErrors.c: In function ‘createA’:
src/cppCheck/CheckErrors.c:34:4: warning: function returns address of local variable [-
Wreturn-local-addr]
    return a;
    ^
src/cppCheck/CheckErrors.c: In function ‘destroyA’:
src/cppCheck/CheckErrors.c:40:8: warning: unused variable ‘z’ [-Wunused-variable]
    int z = 10;
        ^
src/cppCheck/CheckErrors.c: In function ‘a’:
src/cppCheck/CheckErrors.c:58:9: warning: variable ‘b’ set but not used [-Wunused-but-s
et-variable]
    char b[20];
         ^
compiling Factorial.c
```

Figure 4.30: Compilation results

```
Tiago@Tiago-PC ~/sampleC/test
$ cppcheck --enable=all --suppress=missingIncludeSystem -I ./include/cppCheck/ ./src/cppCheck/
Checking src/cppCheck/CheckErrors.c...
[src/cppCheck/CheckErrors.c:22]: (style) The scope of the variable 'i' can be reduced.
[src/cppCheck/CheckErrors.c:25]: (style) Variable 'i' is assigned a value that is never used.
[src/cppCheck/CheckErrors.c:33]: (style) Variable 'a' is not assigned a value.
[src/cppCheck/CheckErrors.c:40]: (style) Variable 'z' is assigned a value that is never used.
[src/cppCheck/CheckErrors.c:63]: (style) Variable 'b' is assigned a value that is never used.
[src/cppCheck/CheckErrors.c:34]: (error) Pointer to local array variable returned.
[src/cppCheck/CheckErrors.c:59]: (error) Array 'a[10]' accessed at index 10, which is out of bounds.
[src/cppCheck/CheckErrors.c:63]: (error) Buffer is accessed out of bounds: a
[src/cppCheck/CheckErrors.c:63]: (error) Array 'a[10]' accessed at index 19, which is out of bounds.
[src/cppCheck/CheckErrors.c:41]: (error) Memory pointed to by 'p' is freed twice.
Checking usage of global functions..
[src/cppCheck/CheckErrors.c:12]: (style) The function 'CheckErrors_Create' is never used.
[src/cppCheck/CheckErrors.c:16]: (style) The function 'CheckErrors_Destroy' is never used.
[src/cppCheck/CheckErrors.c:54]: (style) The function 'a' is never used.
[src/cppCheck/CheckErrors.c:45]: (style) The function 's' is never used.
[src/cppCheck/CheckErrors.c:20]: (style) The function 'z' is never used.
```

Figure 4.31: Cppcheck analysis result from the code available in Annex B

After analyzing the above results, it is possible to create a table comparing both, see Table 4.4. The ***x*** mark indicates if the issue was found by Cppcheck or/and the GCC compiler. It is

possible to conclude that Cppcheck can detect more variety of useful issues then the compiler, not only style issues but errors in code.

ISSUE	COMPILER	CPPCHECK
Variable ***i*** not used.	x	x
Variable ***z*** not used.	x	x
Variable ***b*** not used.	x	x
Variable ***a*** not initialized.		x
CheckErrors_Create is never used.		x
CheckErrors_Destroy is never used.		x
Function ***a*** is never used.		x
Function ***s*** is never used.		x
Function ***z*** is never used.		x
Scope of variable ***i*** can be reduced.		x
createA returns local variable.	x	x
a[10] out of bounds.		x
a[19] out of bounds.		x
Memory freed twice.		x

Table 4.4: Comparative results between GCC compiler and Cppcheck

b. Copy/Paste Detector

The tool copy/paste detector (CPD) is included on the PMD software, a code analyzer like Cppcheck, but only for Java language. To be able to use CPD it is necessary to download the PMD latest binary distribution. At the moment, PMD is in the 4.2.4 version.

CPD has no installation process, it is just executed by using the PMD ***jar*** files in Cygwin or in a Linux shell. The command-line to run it can be seen in Figure 4.32.

```
java -classpath /path/to/pmdbin-4.2.4/lib/pmd-4.2.4.jar
net.sourceforge.pmd.cpd.CPD --minimum-tokens 10 --language c

--format net.sourceforge.pmd.cpd.XMLRenderer

--files /path/to/files/to/analyze

> /path/to/save/output/file.xml
```

Figure 4.32: Command line to execute CPD

The Java argument ***net.sourceforge.pmd.cpd.CPD*** executes the CPD tool. As in Cppcheck, the command in the figure scans the code and write the result in a ***XML*** file to be used in a CI server. According to the online documentation [59], I created the Table 4.5 describing each option used in the execution command. The minimum token length that should be reported as

duplicate code must be adapted to a reasonable value, depending on the level of duplicated code intended to find.

OPTION	MEANING
`-cp /path/to/pmdbin-4.2.4/lib/pmd-4.2.4.jar`	Class search path for PMD jar file.[8]
`--minimum-tokens 10`	The minimum token length that should be reported as a duplicate.
`--language c`	The language of the source code to scan.
`--format net.sourceforge.pmd.cpd.XMLRenderer`	Define the report file format to ***XML***.
`--files /path/to/files/to/analyse`	List of files and directories to process.
`> /path/to/save/output/file.xml`	Defining the name and location of the output file.

Table 4.5: Options used to execute CPD

CPD also gives the possibility to suppress duplicated code warnings by adding a comment line before and after the piece of code wanted to not throw a warning, as shown in Figure 4.33.

```
@SuppressWarnings("CPD-START")
int function(int x, int y){
// code here will be ignored for the duplication detection
}
@SuppressWarnings("CPD-END”)
```

Figure 4.33: Duplicated code suppress warning

The example used to test CPD are dumb code files with some duplication code and they can be seen in Annex B. The result of the analysis is presented in the Figure 4.34. The results are shown from the large code duplication to the small one. The first duplication warning is about an equal set of variables in two different C files, ***ModuleA*** and ***ModuleB***. This can be solved by removing them from both locations and creating a structure in the header file with these variables. The second duplicated code warning shows that two function calls are equally made in both ***switch*** cases. This can be solved just by moving out of the conditional ***switch*** and replacing this ***switch*** with an ***if…else*** statement. The last duplicated code founded is a false positive, a small code duplication, modifying it is useless. All the refactored code can be seen in the Annex B.

[8] It can be used the option *cp* instead of *classpath*.

```
tfa@tfa-VirtualBox:~/Documents/Report/CPD/Code$ java -classpath ../PMD_
.2.4/lib/pmd-4.2.4.jar net.sourceforge.pmd.cpd.CPD --minimum-tokens 15
e c --files .
Added /home/tfa/Documents/Report/CPD/Code/./ModuleB.c
Added /home/tfa/Documents/Report/CPD/Code/./ModuleA.c
Found a 6 line (46 tokens) duplication in the following files:
Starting at line 7 of /home/tfa/Documents/Report/CPD/Code/./ModuleB.c
Starting at line 15 of /home/tfa/Documents/Report/CPD/Code/./ModuleA.c

        char strOK[5]    = "OK;\n";
        char strNOK[7]    = "NOK;\n";
        char strNA[7]    = "N/A;\n";
        char strAux1[6]    = "Teste";
        char strSimul[11]  = "Simulacao ";
        char buffer[6];
=====================================================================
Found a 4 line (16 tokens) duplication in the following files:
Starting at line 18 of /home/tfa/Documents/Report/CPD/Code/./ModuleB.c
Starting at line 24 of /home/tfa/Documents/Report/CPD/Code/./ModuleB.c

                case 2:
                        type(strAux1);
                        send(strOK);
                        result = aaa(var+1024);
=====================================================================
Found a 4 line (15 tokens) duplication in the following files:
Starting at line 17 of /home/tfa/Documents/Report/CPD/Code/./ModuleB.c
Starting at line 22 of /home/tfa/Documents/Report/CPD/Code/./ModuleA.c

        switch(abc) {
                case 1:
                        type(strAux1);
                        send(strOK);

tfa@tfa-VirtualBox:~/Documents/Report/CPD/Code$
```

Figure 4.34: CPD analysis result from code in Annex C

It is possible to conclude that the key to use CPD is the value of the minimum tokens to be considered as duplicated code. If the value is small, is more likely to found false positives, if not, it is almost certain that the results will have duplicated code that should be refactored in order to reduce complexity and improve the code readability and the source code maintainability.

4.4 Continuous Integration Server

The objective of having a CI server is to create a build to execute the unit tests of all developers and to run the static analysis tools. These tools will scan for errors, for code poorly structured, for duplicated code, warnings and open tasks left. When new code is submitted to the SCM system, everything will be executed during the Jenkins build. The results will be sent to developers and will be available with more detail on the server.

The CI server chosen, Jenkins, can be used trough the java web archive (.war), or the native package. The second one installs Jenkins as an OS service, which starts Jenkins automatically whenever Windows or Linux is restarted. The first one executes Jenkins as an application, it only be online if someone execute it. Having Jenkins always running is the recommended option to use, because it makes possible to have builds and results any time.

When Jenkins is fully up, configured and running, first of all it is necessary to install specific plugins. They will show trend reports with all the information about the unit testing results, for example. These plugins will take the XML report files generated and show all the information in the file in a more useful way. There are other plugins to enhance the Jenkins experience that will be also taken into account.

After creating a new Jenkins item/job[9] and configure it with the essential aspects, for example choosing Git as the SCM system, the next step is to build the project. Jenkins can execute multiple shell commands sequentially, calling them as build steps. The static analyze tools and the unit tests will be executed in build steps (see Figure 4.35).

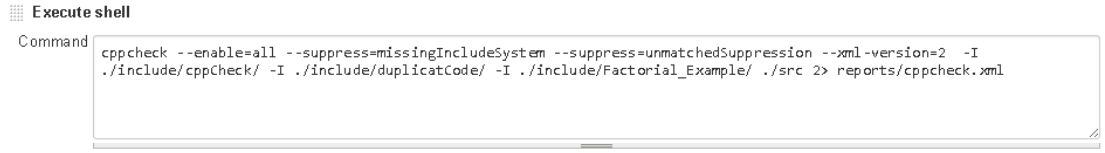


Figure 4.35: Example of a Jenkins build step to execute Cppcheck

There are also the post build actions, executed after the build steps, where Jenkins plugins will take the ***XML*** files and generate the results to be presented on the server. The post build actions will execute the needed Jenkins plugins. Normally there is a specific plugin for every static tool and for xUnit tests. In Jenkins, it is necessary to install ***xUnit*** plugin [70], to have the results for unit testing. For the static analysis tools it must be installed the ***Cppcheck*** plugin, ***Dry*** plugin [71] for the Copy/Paste Detector tool and finally the ***warnings*** and ***task scanner*** plugins. In addition to this list, there are other plugins to enhance the Jenkins experience: ***Email-ext*** plugin, wich allows the configuration of every aspect of email notifications and ***Claim*** plugin to allow users to take responsibility for failed builds [72].

After building an item/job on Jenkins, there is a build result state that changes if something goes wrong when executing the build steps or due to the plugins configurations (see Table 4.6). For example, if I add a threshold for the number of failed test in the ***xUnit*** plugin and in the CppUTest results the failed tests reached that threshold, then the build will certainly fail.

[9] Item/job can be associated with the concept of project.

Build Status	Icon	Meaning
Stable		Jenkins built successfully and no plugins report it as unstable or failed.
Failed		The building process was not able to finish due to an error or a plugin changed the build status to failed.
Unstable		The build was built successfully and one or more plugins report it unstable.

Table 4.6: Jenkins build status

With the information about the number of failed, unstable and successful builds, each item/job creates a health report named ***build stability*** (see Table 4.7) which can vary between three states, ***sunny***, ***cloudy*** and ***thunder*** (0% to 100%). Some plugins have also a health report which influences the item/job health report. For example, the plugin Cppcheck has a health report created based on the results of the static analysis tool with the same name.

Item/job health	Description
	No recent builds failed.
	40-60% of builds failed.
	All the recent builds failed.

Table 4.7: Jenkins build stability description

The objective now is to configure the Jenkins plugins to read the output files from the tools and report the results. These plugins can present results with trend graphs, tables and other type of organized data that will be a lot easier and faster to developers understand.

xUnit

Jenkins has a default plugin to report **junit** files, but ***xUnit*** has proven to be better, with more useful options. The name of the ***XML*** output files created by CppUTest always have the prefix ***cpputest_*** making easy the identification of all (see Figure 4.36). The configuration of this plugin allows the definition of how many failed/skipped tests is necessary to fail the build or make it unstable, unlike the default Junit plugin where this is not possible.

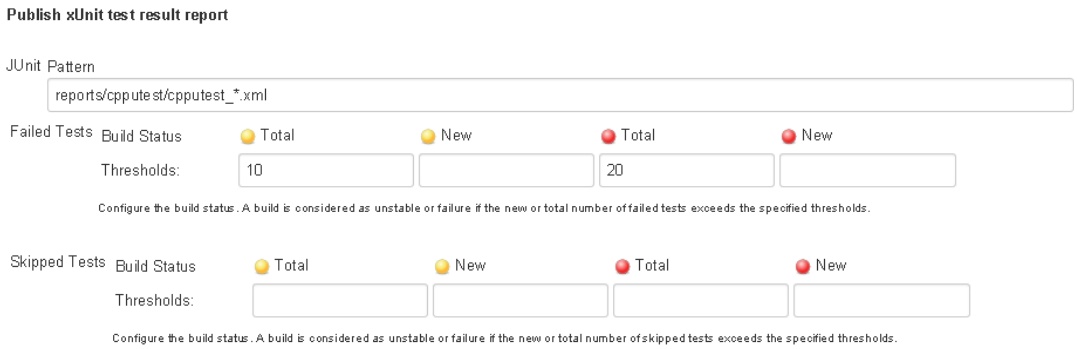


*Figure 4.36: An example of the **xUnit** plugin configuration*

Cppcheck

The plugin configuration is similar to the last one, the differences are the possibility to change the project health, and the possibility to choose which kind of Cppcheck problems encountered will count to change the build state (see Figure 4.37).

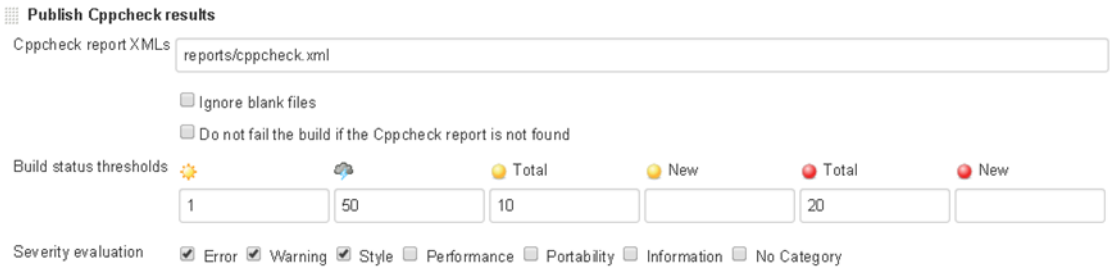


*Figure 4.37: An example of the **Cppcheck** plugin configuration*

DRY

Dry plugin takes the CPD results and categorize it in three priorities, high, normal and low. The low category value is defined in minimum tokens on the command-line execution, the other two categories are configured on the plugin configuration (Figure 4.38). As in the other plugins, there is a number of duplicated warnings, in each category, that change the build state to unstable/failed as well as the configuration of the plugin health report.

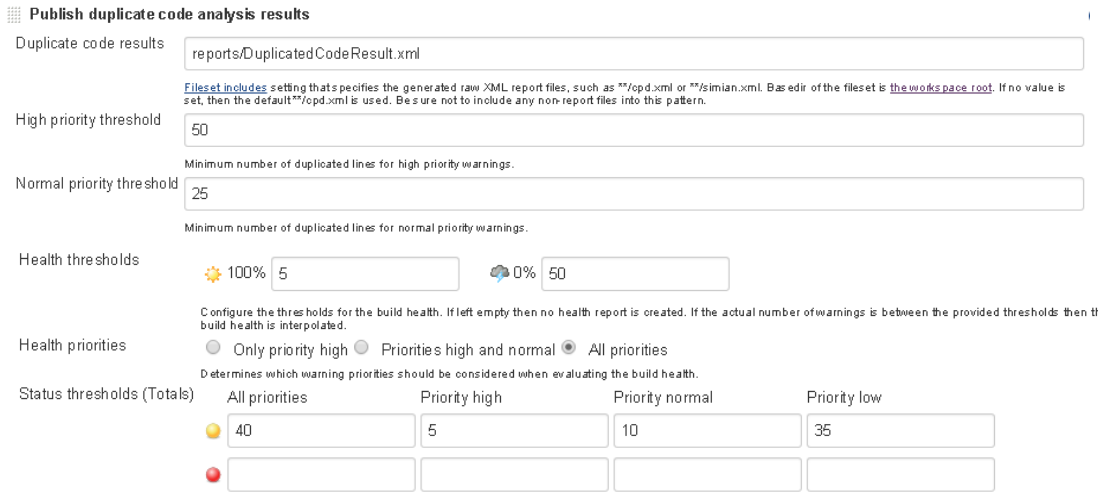


*Figure 4.38: An example of the **dry** plugin configuration*

Warnings

This plugin only searches for warnings in the console log. The configuration is only about defining which is the compiler to scan, in this case is the GCC C/C++ compiler.

Task Scanner

Open tasks normally are represented as specific comments on the code, as such ***TODO*** or ***FIXME***, but they can be different for each development team. This plugin allows the possibility to choose which name of the open tasks to scan for.

Email-ext

In addtion to the Jenkins plugins to do static analysis, there are more plugins available to have a better experience with Jenkins, one of them is the detailed feedback given by email. The plugin is named email-ext [73] and can be trigered to send to different mail addresses according to the state of build, if failed, if changed to unstable, among others. The email content can be configured, it is possible to use Jelly and Groovy scripts. it is possible to customize when an email is sent, who should receive it, and what the email says [73];

Claim plugin

When a Jenkins build fails or changes the project state to unstable, this plugin allows the developers to go on Jenkins page and claim the build to inform the rest of the team that someone is fixing it.

To conclude, Figure 4.39 shows what happens inside the continuous server, what are the steps to build a Jenkins project. In short, Jenkins will check out the code on the SCM system, run all the build steps, create the results in the post build actions and send feedback to developers.

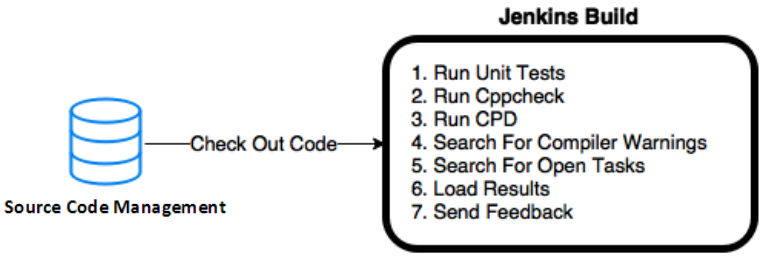


Figure 4.39: Jenkins build steps

When Jenkins has an item/job configured it will be able to build for every change in the main repository. The results will tell if the code is clean. For example, during a product development, Jenkins build result shows that some tests failed. To fix them, a developer creates a new branch named ***fixFailedTests*** (see Figure 4.40). After fixing the tests, he commits the code locally, pushes the branch to the main repository and triggers a Jenkins build. The server will build the last modified branch, the ***fixFailedTests***. After a successful build, the developer can safely merge the branch with the ***master***, where the main source is.

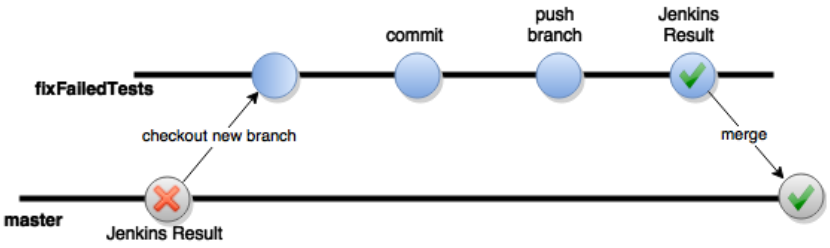


Figure 4.40: Flow to fix failed tests

4.5 Summary

The new solution can be seen in Figure 4.41 which demonstrates a workflow where is possible to use the agile practices, test driven development and continuous integration. The TDD practice is presented on the local and server unit testing. With the CppUTest framework integrated on Atmel Studio is possible to write code, tests and execute them on the same development target for every modification in code. The CI practice is presented by joining Git with Jenkins. With Git, it is possible to create branches for each feature and push it to the main repository and Jenkins analyses the feature branch for issues before merging it with the main branch.

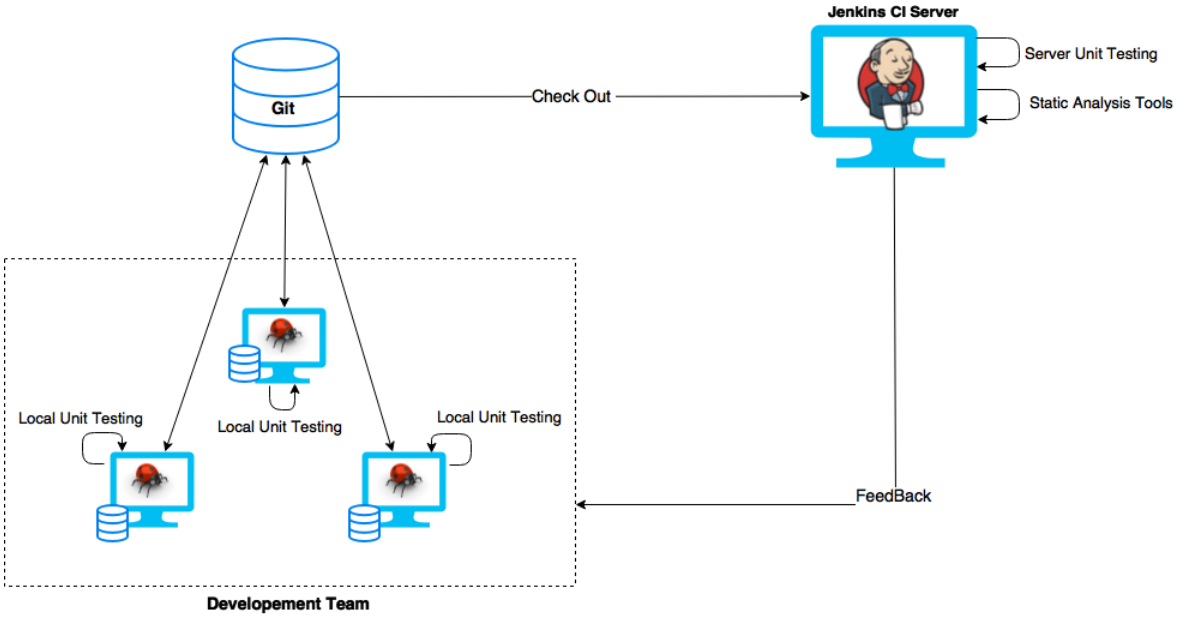


Figure 4.41: Final workflow for the embedded software development

chapter 5

Results and Validation

So far, I have explained how to configure an environment to allow CI and TDD. Now is time to apply these solutions to some projects and test the tools used for it. We have seen examples of usage for Cppcheck, CPD and CppUTest independently (see chapter 4). Now, using a virtual machine with Ubuntu 14.04, I will simulate a CI server with Jenkins and use it for two embedded projects, the one from Exatronic and the other from James Greening book.

a. Led Driver Project

The first project to apply the methodology suggested in this dissertation will be the one from James Greening book [63], a LED driver. This driver is for a system that uses LEDs to communicate status to users. I simulate the development using Atmel studio project for an ***Atmega32*** device. Initially, the code used a stub to fake the LEDs address, a simple variable on the code, but I replaced it for a device register, ***PORTC***. The project structure in Atmel studio can be seen in Figure 5.1.

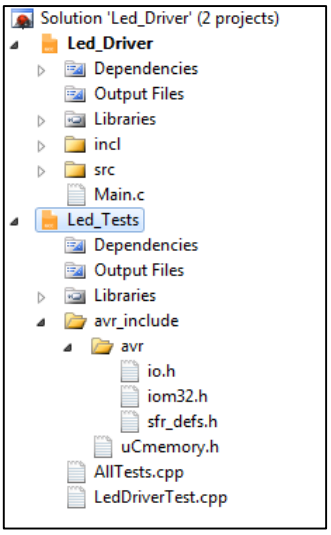


Figure 5.1: Led_Driver project structure

After building the project, some tests failed, as shown in Figure 5.2. After verifying the expected values in the output window (see Figure 5.3), it was obvious that ***Led_Driver*** was working for registers with 16-bits, so I changed the code to use only 8-bits. I was able to run unit tests and change code to pass them on the same platform, all in Atmel studio. After making the tests pass, the project compiled with no errors as shown in Figure 5.4.

5 Errors | 2 Warnings | 0 Messages

	Description	File	Line	Co...	Project
3	Failure in TEST(LedDriver, AllOn)	LedDriverTest.c	184	1	Led_Tests
4	Failure in TEST(LedDriver, OutOfBoundsTurnOffDoesNoHarm)	LedDriverTest.c	133	1	Led_Tests
5	Failure in TEST(LedDriver, UpperAndLowerBounds)	LedDriverTest.c	107	1	Led_Tests
6	Failure in TEST(LedDriver, TurnOffAnyLed)	LedDriverTest.c	89	1	Led_Tests

*Figure 5.2: **Led_Driver** Failed Tests*

```
.....
C:\Users\Tiago\Desktop\Nova pasta\Led_Driver\Led_Tests\LedDriverTest.cpp(133,1): error: Failure in
TEST(LedDriver, OutOfBoundsTurnOffDoesNoHarm)
            expected <65535 0xffff>
            but was  <  255 0x00ff>
        ..
C:\Users\Tiago\Desktop\Nova pasta\Led_Driver\Led_Tests\LedDriverTest.cpp(107,1): error: Failure in
TEST(LedDriver, UpperAndLowerBounds)
            expected <32769 0x8001>
            but was  <    1 0x0001>
        ..
C:\Users\Tiago\Desktop\Nova pasta\Led_Driver\Led_Tests\LedDriverTest.cpp(89,1): error: Failure in
TEST(LedDriver, TurnOffAnyLed)
            expected <65407 0xff7f>
            but was  <  127 0x007f>
```

*Figure 5.3 **Led_Driver** output window after compilation*

```
------ Build started: Project: Led_Tests, Configuration: Debug AVR ------
Build started.
Project "Led_Tests.cproj" (default targets):
Target "PreBuildEvent" skipped, due to false condition; ('$(PreBuildEvent)'!='') was evaluated as (''!='').
Target "CoreBuild" in file "C:\Program Files (x86)\Atmel\Atmel Studio 6.2\Vs\Compiler.targets" from project "C:\Users
\Tiago\Documents\Atmel Studio\6.2\Led_Driver\Led_Tests\Led_Tests.cproj" (target "Build" depends on it):
    Task "RunCompilerTask"
        Shell Utils Path C:\Program Files (x86)\Atmel\Atmel Studio 6.2\shellUtils
        C:\Program Files (x86)\Atmel\Atmel Studio 6.2\shellUtils\make.exe -C "C:\Users\Tiago\Documents\Atmel Studio
\6.2\Led_Driver" -f "Makefile_NoImport" all
        make: Entering directory `C:/Users/Tiago/Documents/Atmel Studio/6.2/Led_Driver'
        Compiling AllTests.cpp
        g++ -D__AVR_ATmega32__ -DCPPUTEST -IC:\Users\Tiago\Desktop\Local_Test\cpputest/include -include C:\Users
\Tiago\Desktop\Local_Test\cpputest/include/CppUTest/MemoryLeakDetectorNewMacros.h  -ILed_Tests  -ILed_Driver/incl  -
ILed_Driver/mocks  -ILed_Tests/avr_tests -c Led_Tests/AllTests.cpp -o objs/Led_Tests/AllTests.o
        Compiling LedDriverTest.cpp
        g++ -D__AVR_ATmega32__ -DCPPUTEST -IC:\Users\Tiago\Desktop\Local_Test\cpputest/include -include C:\Users
\Tiago\Desktop\Local_Test\cpputest/include/CppUTest/MemoryLeakDetectorNewMacros.h  -ILed_Tests  -ILed_Driver/incl  -
ILed_Driver/mocks  -ILed_Tests/avr_tests -c Led_Tests/LedDriverTest.cpp -o objs/Led_Tests/LedDriverTest.o
        Compiling LedDriver.c
        gcc -D__AVR_ATmega32__ -DCPPUTEST -IC:\Users\Tiago\Desktop\Local_Test\cpputest/include -std=c99 -include C:
\Users\Tiago\Desktop\Local_Test\cpputest/include/CppUTest/MemoryLeakDetectorMallocMacros.h  -ILed_Tests  -
ILed_Driver/incl  -ILed_Driver/mocks  -ILed_Tests/avr_tests -c Led_Driver/src/LedDriver.c -o objs/Led_Driver/src/
LedDriver.o
        Compiling RuntimeErrorStub.c
        gcc -D__AVR_ATmega32__ -DCPPUTEST -IC:\Users\Tiago\Desktop\Local_Test\cpputest/include -std=c99 -include C:
\Users\Tiago\Desktop\Local_Test\cpputest/include/CppUTest/MemoryLeakDetectorMallocMacros.h  -ILed_Tests  -
ILed_Driver/incl  -ILed_Driver/mocks  -ILed_Tests/avr_tests -c Led_Driver/src/RuntimeErrorStub.c -o objs/Led_Driver/
src/RuntimeErrorStub.o
        objs/Led_Tests/AllTests.o objs/Led_Tests/LedDriverTest.o objs/Led_Driver/src/LedDriver.o objs/Led_Driver/src/
RuntimeErrorStub.o
        Linking Led_Test
        g++ objs/Led_Tests/AllTests.o objs/Led_Tests/LedDriverTest.o objs/Led_Driver/src/LedDriver.o objs/Led_Driver/
src/RuntimeErrorStub.o -D__AVR_ATmega32__ -DCPPUTEST -IC:\Users\Tiago\Desktop\Local_Test\cpputest/include -LC:\Users
\Tiago\Desktop\Local_Test\cpputest/lib  -o Led_Test -lCppUTest -lCppUTestExt -lpthread -lstdc++
        ./Led_Test
        .........!.....
        OK (15 tests, 14 ran, 19 checks, 1 ignored, 0 filtered out, 3 ms)
        make: Leaving directory `C:/Users/Tiago/Documents/Atmel Studio/6.2/Led_Driver'
    Done executing task "RunCompilerTask".
    Task "RunOutputFileVerifyTask"
```

Figure 5.4: Build output of the Led_Driver solution.

Now, I placed the project in an online repository to simulate a development environment. Jenkins checked out the code and made a build. The result can be seen in Figure 5.5. As expected, there was no test failures, however Cppcheck reported nine issues. Figure 5.6 presents a table with all the results detailed.

*Figure 5.5: **Led_Driver** results on Jenkins*

State	File	Line	Severity	Type	Inconclusive	Message
unchanged	Led_Driver/src/LedDriver.c	33	style	unusedFunction	false	The function 'LedDriver_Create' is never used.
unchanged	Led_Driver/src/LedDriver.c	41	style	unusedFunction	false	The function 'LedDriver_Destroy' is never used.
unchanged	Led_Driver/src/LedDriver.c	114	style	unusedFunction	false	The function 'LedDriver_IsOff' is never used.
unchanged	Led_Driver/src/LedDriver.c	100	style	unusedFunction	false	The function 'LedDriver_TurnAllOff' is never used.
unchanged	Led_Driver/src/LedDriver.c	94	style	unusedFunction	false	The function 'LedDriver_TurnAllOn' is never used.
unchanged	Led_Driver/src/LedDriver.c	85	style	unusedFunction	false	The function 'LedDriver_TurnOff' is never used.
unchanged	Led_Driver/src/LedDriver.c	72	style	unusedFunction	false	The function 'LedDriver_TurnOn' is never used.
unchanged	Led_Driver/src /RuntimeErrorStub.c	49	style	unusedFunction	false	The function 'RuntimeErrorStub_GetLastParameter' is never used.
unchanged	Led_Driver/src /RuntimeErrorStub.c	33	style	unusedFunction	false	The function 'RuntimeErrorStub_Reset' is never used.

*Figure 5.6: Detailed Cppcheck results for **Led_Driver***

The nine ***style*** issues are about unused functions. It is normal to have these amount of unused functions because ***Led_Driver*** is a module to be used by other modules, in this case, this functions were used to make tests.

Apart the Cppcheck results, there were no other issues in the remaining static analysis tools.

b. Mafalda Project

The next embedded project used was the one from Exatronic, named Mafalda. Briefly, Mafalda is the project where tests were made for an Exatronic product about a LED boards adapter for trucks. The tests were made in an evaluation board of the series AVR XMEGA 8-bits, which has a complex processor comparing to the one used for the James Greening LED driver project.

Mafalda has a set of unit tests which uses the CppUTest framework, but they were not used with the TDD practice because they were created for a finished project. However, Cygwin was used to compile and execute tests, while the production code was in the Atmel studio. The objective here was to make Atmel studio ready to compile and execute the unit tests from Mafalda, in order to have only one development target.

First I tried to compile the available tests, but I found myself trying to load all the project with all the dependencies just to test some files. So, I tried to understand which were the files under test and their dependencies. I found a chain of dependencies for the code under test, but not all were necessary to run the unit tests. Figure 5.7 shows almost all dependencies, where the gray box is the code under test. In order to compile the unit tests from Mafalda, I had to isolate the code from unnecessary dependencies as presented in Figure 5.8. The dependency from the Atmel software framework header file was removed.

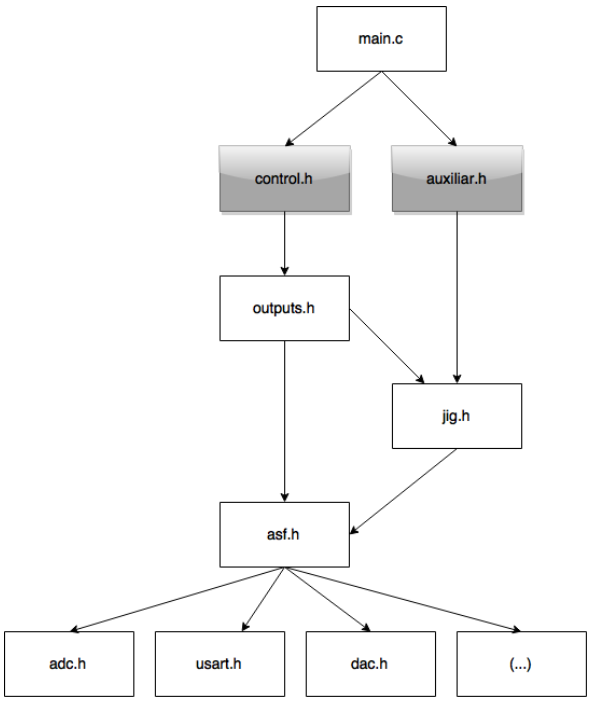


Figure 5.7: Main code dependencies

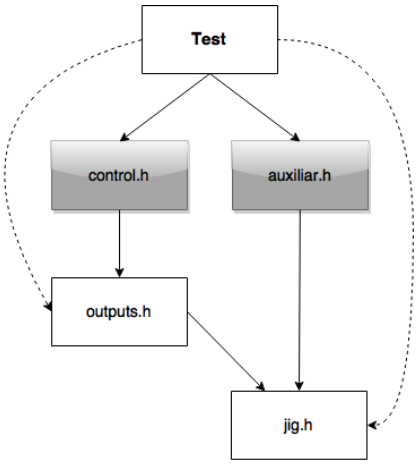


Figure 5.8: Isolating Mafalda from unnecessary dependencies

There are two ways to isolate code in order to compile the unit tests: creating a new file similar to the original, but without the unnecessary code and header files. James Greening in his book named this file as ***Test Double;*** using macros to exclude the unwanted code and header files. After that isolating the code, I was able to put the unit tests inside the Atmel studio with the production code and get some results (see Figure 5.9).

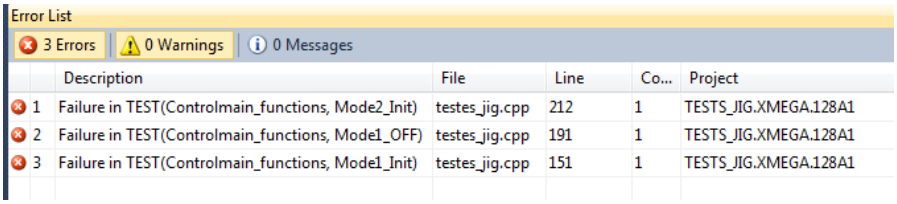


Figure 5.9: First result of Mafalda on Atmel Studio

Whiteout losing the reliability of code and tests, I modified some code to finally see all tests from Mafalda passing inside the Atmel studio. The structure from the project solution can be seen in Figure 5.10.

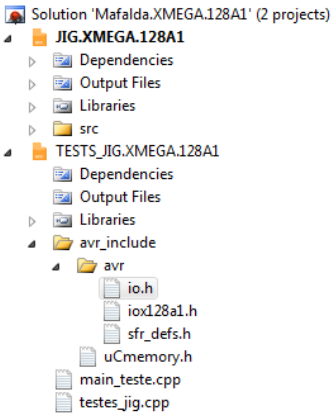


Figure 5.10 Mafalda project structure

The Atmel system files needed for testing were ***io***, ***sfr_defs*** and the respective header device, the ***iox128a1***. The structure to do unit testing is always similar, after doing some unit testing it is easy to understand and memorize what is necessary for each project. The test compilation result from Atmel studio is presented in Figure 5.11.

Figure 5.11: Mafalda testing result

Now I will show the results from the Jenkins server in Figure 5.12. There are no failed tests, as expected, however there are 78 warnings for duplicated code, one open task left and 24 Cppcheck issues.

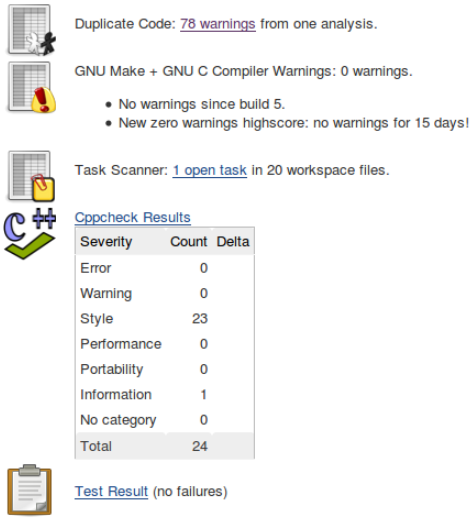


Figure 5.12: Jenkins build result for Mafalda

Starting with the results from CPD: the value for the minimum tokens was 50, the normal priority threshold was 25 duplicated lines and for the high priority was 50 lines. According to this values, the CPD generated the tables in Figure 5.13.

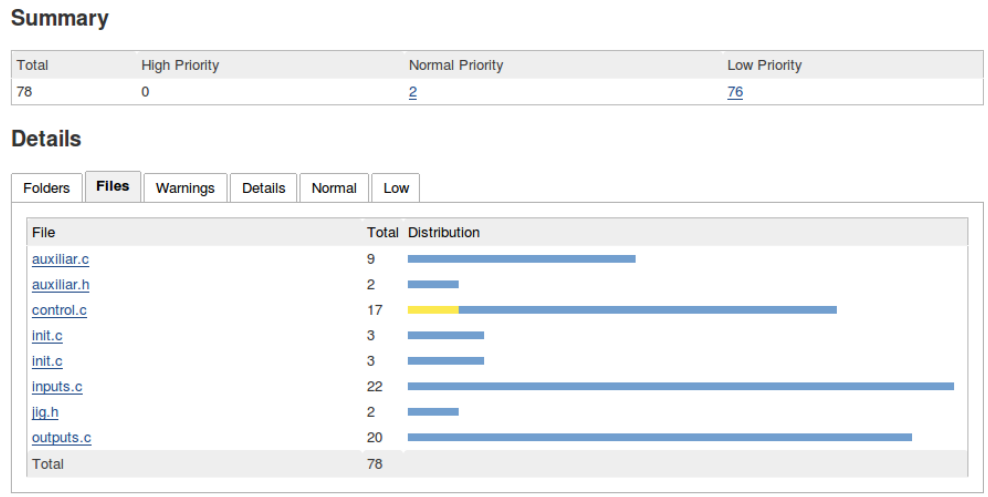


Figure 5.13: CPD detailed results

There is a warning in the file ***control.c*** with more than 25 lines of duplicated code (the yellow part of the line in Figure 5.13). Inside a conditional ***switch*** there was two cases with almost the same code, as shown in Figure 5.14. An example of a solution could be using a function with all the duplicated code inside and an ***if*** statement to call the method ***toggle_24V(LIGHT_RIGHT_STOP)*** for each case.

Another example is a low warning with 5 duplicated lines in the file with high rate of duplicated code, ***inputs.c.*** As shown in Figure 5.15, all the ***if*** statements have the same code, the only thing that changes is the ***ADC*** register. To solve this and other warnings it would be necessary to replace the code inside the conditional statement into a function with the register as argument and convert all the ***ifs*** statements into a ***switch***.

```
089     case MODE_FLASHERS_STOP_ON:
090     {
091       if(jig.initMode == TRUE)
092       {
093         /*
094         set_bulb_lamp(LIGHT_RIGHT_STOP, ON);
095         set_bulb_lamp(LIGHT_LEFT_STOP, ON);
096         set_bulb_lamp(LIGHT_RIGHT_FLASHER, ON);
097         set_bulb_lamp(LIGHT_LEFT_FLASHER, OFF);
098         */
099
100         //Init loads
101         set_led_lamp(LIGHT_RIGHT_STOP,LED_70mA,ON);
102         set_led_lamp(LIGHT_LEFT_STOP,LED_70mA,ON);
103         set_led_lamp(LIGHT_RIGHT_FLASHER,LED_70mA,ON);
104         set_led_lamp(LIGHT_LEFT_FLASHER,LED_70mA,ON);
105
106         //Set loads
107         CONTROLPORT2.OUTSET = LOADSTOPMINR_MASK;
108         CONTROLPORT1.OUTSET = LOADSTOPMINL_MASK;
109         CONTROLPORT1.OUTSET = LOADRIGHTMIN_MASK;
110         CONTROLPORT1.OUTSET = LOADLEFTMIN_MASK;
111
112         //Set power
113         set_24V(LIGHT_RIGHT_STOP, ON);
114         set_24V(LIGHT_RIGHT_FLASHER, OFF);
115         set_24V(LIGHT_LEFT_FLASHER, ON);
116
117         jig.initMode = FALSE;
118       }
119
120       if (jig.cCiclosToggle == TOGGLE_FREQ)
121       {
122         jig.cCiclosToggle = 0;
123         //togglebulb(LIGHT_RIGHT_FLASHER);
124         //togglebulb(LIGHT_LEFT_FLASHER);
125         set_buzzer(OFF);
126
127         toggle_24V(LIGHT_RIGHT_FLASHER);
128         toggle_24V(LIGHT_LEFT_FLASHER);
129       }
130     }
131     break;
132
133
```

```
134     case MODE_FLASHERS_STOP_TGL:
135     {
136       if(jig.initMode == TRUE)
137       {
138         /*
139         set_bulb_lamp(LIGHT_RIGHT_STOP, ON);
140         set_bulb_lamp(LIGHT_LEFT_STOP, ON);
141         set_bulb_lamp(LIGHT_RIGHT_FLASHER, ON);
142         set_bulb_lamp(LIGHT_LEFT_FLASHER, OFF);
143         */
144
145         //Init loads
146         set_led_lamp(LIGHT_RIGHT_STOP,LED_70mA,ON);
147         set_led_lamp(LIGHT_LEFT_STOP,LED_70mA,ON);
148         set_led_lamp(LIGHT_RIGHT_FLASHER,LED_70mA,ON);
149         set_led_lamp(LIGHT_LEFT_FLASHER,LED_70mA,ON);
150
151         //Set loads
152         CONTROLPORT2.OUTSET = LOADSTOPMINR_MASK;
153         CONTROLPORT1.OUTSET = LOADSTOPMINL_MASK;
154         CONTROLPORT1.OUTSET = LOADRIGHTMIN_MASK;
155         CONTROLPORT1.OUTSET = LOADLEFTMIN_MASK;
156
157         //Set power
158         set_24V(LIGHT_RIGHT_STOP, ON);
159         set_24V(LIGHT_RIGHT_FLASHER, OFF);
160         set_24V(LIGHT_LEFT_FLASHER, ON);
161
162         jig.initMode = FALSE;
163       }
164
165       if (jig.cCiclosToggle == TOGGLE_FREQ)
166       {
167         jig.cCiclosToggle = 0;
168         //togglebulb(LIGHT_RIGHT_STOP);
169         //togglebulb(LIGHT_LEFT_STOP);
170         //togglebulb(LIGHT_RIGHT_FLASHER);
171         //togglebulb(LIGHT_LEFT_FLASHER);
172         set_buzzer(OFF);
173
174         toggle_24V(LIGHT_RIGHT_STOP);
175         toggle_24V(LIGHT_RIGHT_FLASHER);
176         toggle_24V(LIGHT_LEFT_FLASHER);
177       }
178     }
179     break;
```

*Figure 5.14: Duplicated code with more than 25 lines in the file **control.c***

```
405   if (lamp_id == LIGHT_RIGHT_STOP)
406   {
407     set_short_simulation(lamp_id, ON);
408     adc_start_conversion(&ANALOGPORT1, ADC_CH0IF_bp);
409     jig.adcValuesArray[jig.currentState] = adc_get_unsigned_result(&ANALOGPORT1, ADC_CH0IF_bp) & 0x0fff;
410     set_short_simulation(lamp_id, OFF);
411   }
412
413   if (lamp_id == LIGHT_LEFT_STOP)
414   {
415     set_short_simulation(lamp_id, ON);
416     adc_start_conversion(&ANALOGPORT1, ADC_CH1IF_bp);
417     jig.adcValuesArray[jig.currentState] = adc_get_unsigned_result(&ANALOGPORT1, ADC_CH1IF_bp) & 0x0fff;
418     set_short_simulation(lamp_id, OFF);
419   }
420
421   if (lamp_id == LIGHT_LEFT_FLASHER)
422   {
423     set_short_simulation(lamp_id, ON);
424     adc_start_conversion(&ANALOGPORT1, ADC_CH2IF_bp);
425     jig.adcValuesArray[jig.currentState] = adc_get_unsigned_result(&ANALOGPORT1, ADC_CH2IF_bp) & 0x0fff;
426     set_short_simulation(lamp_id, OFF);
427   }
428
429   if (lamp_id == LIGHT_RIGHT_FLASHER)
430   {
431     set_short_simulation(lamp_id, ON);
432     adc_start_conversion(&ANALOGPORT1, ADC_CH3IF_bp);
433     jig.adcValuesArray[jig.currentState] = adc_get_unsigned_result(&ANALOGPORT1, ADC_CH3IF_bp) & 0x0fff;
```

*Figure 5.15: Duplicated code inside of all the **if…else** statements*

The results from the open tasks scanner are present in Figure 5.16. The configuration was configured to find open tasks with the name ***TODO*** or ***FIXME***, the more common ones. The scanner found one open task in the file ***init.c*** and the associated message (see Figure 5.17).

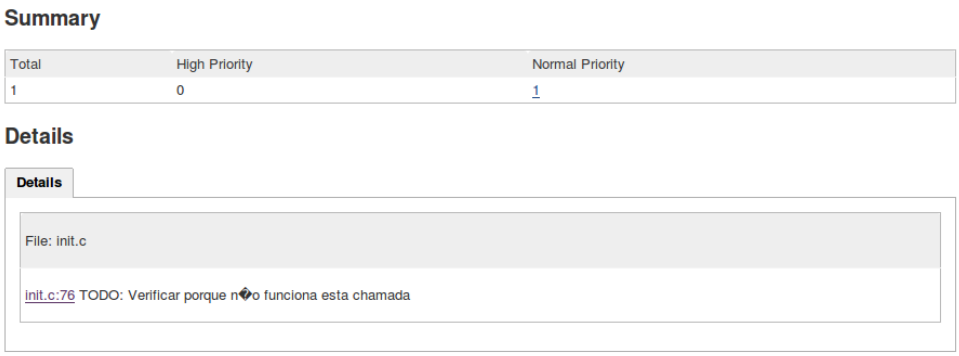


Figure 5.16: Open task scanner result

```
073 void dac_init(void)
074 {
075
076   //sysclk_enable_peripheral_clock(&DACB);                    //TODO: Verificar porque n@o funciona esta chamada
077   sysclk_enable_module(SYSCLK_PORT_B, SYSCLK_DAC);
078
079   dac_read_configuration(&DACPORT, &jig.dacConfig);
```

Figure 5.17: Line with an open task

Finally, the results from Cppcheck can be seen in Figure 5.18. It only mentions style issues about unused variables and functions. Mafalda is a project to test another, so it is normal to have some unused thing because they are being used in the unit tests or in the project tested by Mafalda.

About the ***information*** issue, Cppcheck can scan header files, but if is a system header it won't be necessary to scan. With the Mafalda project I saw that some system headers were included as simple headers and not as system headers. This is why the information issue appeared. The system headers should not be included in the command-line to execute Cppcheck, it is code that is supposed to be correct. The only header files that must be included are the ones created by the project developers. Figure 5.18 shows an information about which are the missing include files. I add the argument ***--check-config*** in command-line execution of Cppcheck to see which are they. The result, as shown in Figure 5.19, mentions the missing of the file ***board.h***, a system header included by ***init.c*** as a normal header file.

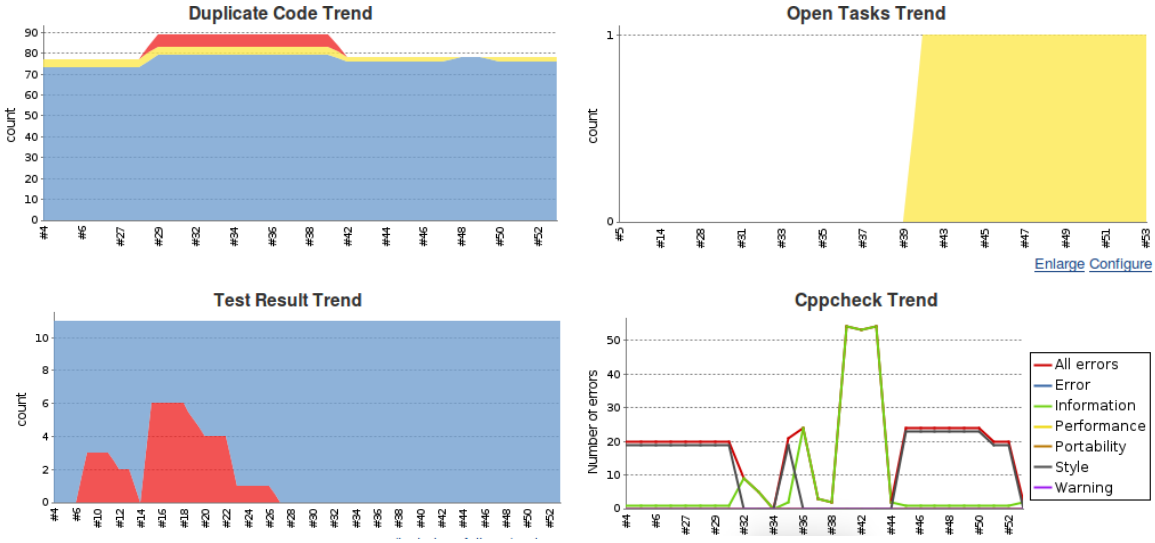
State	File	Line	Severity	Type	Inconclusive	Message
unchanged	original/auxiliar.c	108	style	unusedVariable	false	Unused variable: mean_value_volt
unchanged	original/auxiliar.c	109	style	unusedVariable	false	Unused variable: mean_value_corr
unchanged	original/main.c	26	style	unusedVariable	false	Unused variable: strNS
unchanged	original/main.c	28	style	unreadVariable	false	Variable 'ptrStrNS' is assigned a value that is never used.
unchanged	original/auxiliar.c	80	style	unusedFunction	false	The function 'controlcycle' is never used.
unchanged	original/inputs.c	478	style	unusedFunction	false	The function 'mean_calc' is never used.
unchanged	original/inputs.c	216	style	unusedFunction	false	The function 'read_currents_task' is never used.
unchanged	original/inputs.c	359	style	unusedFunction	false	The function 'read_load_current' is never used.
unchanged	original/inputs.c	403	style	unusedFunction	false	The function 'read_short_current' is never used.
unchanged	original/auxiliar.c	37	style	unusedFunction	false	The function 'read_string' is never used.
unchanged	original/outputs.c	335	style	unusedFunction	false	The function 'set_led1_green' is never used.
unchanged	original/outputs.c	348	style	unusedFunction	false	The function 'set_led2_red' is never used.
unchanged	original/outputs.c	361	style	unusedFunction	false	The function 'set_led3_green' is never used.
unchanged	original/auxiliar.c	105	style	unusedFunction	false	The function 'test_functions' is never used.
unchanged	original/inputs.c	14	style	unusedFunction	false	The function 'test_primary_task' is never used.
unchanged	original/control.c	230	style	unusedFunction	false	The function 'togglebulb' is never used.
unchanged	original/inputs.c	163	style	unusedFunction	false	The function 'test_secondary_task' is never used.
unchanged	original/asf/xmega/boards/jig/init.c	145	style	unusedFunction	false	The function 'wdt_init' is never used.
unchanged	original/auxiliar.c	50	style	unusedFunction	false	The function 'update_clock' is never used.
unchanged			information	missingInclude	false	Cppcheck cannot find all the include files (use --check-config for details)

Figure 5.18: Detailed Cppcheck results

State	File	Line	Severity	Type	Inconclusive	Message
new	original/asf/xmega/boards/jig/init.c	11	information	missingInclude	false	Include file: "board.h" not found.
new	original/asf/xmega/boards/xplain/init.c	43	information	missingInclude	false	Include file: "board.h" not found.

Figure 5.19: Warning about missing include files

The following graphics are the variations of the results while I was making modifications in code and tests. This can be an example of how the results in a product development may vary. The build 53 is the last build made and used to show the previous results.



Besides the feedback given trough the server, Jenkins is also able to send feedback by email after each build made or in accordance with the ***email-ext*** plugin configurations. Figure 5.20 is an example of an email about the build results of Mafalda. It shows the main information like the branch built, the repository changes, static tools and tests results, along with other useful information.

BUILD SUCCESS

Build URL http://localhost:8080/job/Mafalda/51/
Project: Mafalda
Date of build: Mon, 15 Jun 2015 05:27:05 +0100
Build duration: 6.4 sec
Build cause: Started by user anonymous
Build description:
Built on: master

Health Report

Description	Score
Test Result: 0 tests failing out of a total of 11 tests.	100
Build stability: No recent builds failed.	100

Changes

main e test

by tiagofreitas@ua.pt
add /original/main.c
add /original/main.h
add /original/test.c
add /original/test.h

JUnit Tests

Package	Failed	Passed	Skipped	Total
(root)	0	11	0	**11**

Static Analysis Results

Name	Total	High	Normal	Low
Duplicate Code	78	0	2	76
GNU Make + GNU C Compiler Warnings	0	0	0	0
Compiler Warnings	0	0	0	0
Open Tasks	1	0	1	0

CppCheck Result

Error	Warning	Style	Performance	Portability	Information	No Category	Total
0	0	19	0	0	1	0	20

Console Output

Figure 5.20: Email sent by Jenkins using a template created with a jelly script

chapter 6

Conclusion

The agile development may be associated most of the time associated with the development of information systems. But it is possible to take advantage of the agile practices in other types of development, including the embedded one. At the beginning of this dissertation, we chose which agile practices could better fit in an embedded development and divide them in two groups, management and code practices. The management practices can really help a development team to organize itself, to be more focused on the product features to develop, and consequently more productive. The recommended code practices can make a difference on the code development, it may improve the quality of the code and prevent errors. Two of the code agile practices were selected for a more detailed approach, the test driven development and continuous integration practices. The main objective was to create a solution that would allow the implementation of both practices on a company like Exatronic.

In order to have test driven development, I presented a way to write unit tests, develop code and compile them in the same development target, the Atmel studio, using the CppUTest framework. Developers will no longer lose focus and time for having to change to other application to run tests in every code modification. The solution makes easy the adaptation of the practice.

The environment to allow continuous integration needs a server and we chose Jenkins to build all unit tests and tools for every modification in the main repository. The tools executed on the build make static analysis on the code. Cppcheck scans for undetected issues on the compilation process, it can be an issue about style or a critical error. Copy/Paste Detector searches for duplicated code in one file or between one and more. The results encourages the

refactoring practice to improve the readability of the code and to be easier to maintain. During a Jenkins build, there is also a plugin that scans for warnings and other that scans code files for open tasks left. Executing tests and doing static analysis will improve the quality of the code. All these tools and tests could be executed independently of Jenkins, the difference is that using Jenkins is having a place where they are all automatically executed and where the results are all organized in detailed tables and graphics to be shown to developers. Jenkins works like a barrier that separates the developed code from the final code, informing the development team about errors found.

In addition, it was recommended to use Git, a different SCM system from the one used in Exatronic. The mix between the centralized and the feature branch workflow, and the robust branch and merge model, adds a new way to manage the source code that does not deviate from the agile idea.

The presented solution on this dissertation was tested using some examples. For instance, in the project from Exatronic it was possible to see some results on Jenkins that shown the advantages of using the static analysis tools. Some issues could have been noticed right after the implementation and the creation of unit tests could have been faster and simpler using only the Atmel studio for it.

To conclude, this dissertation contributed with practical recommendations to adopt the test driven development and continuous integration practices in the embedded software development cycle.

6.1 Future Work

In the context of the test driven development practice, there is the notion of mocks to be deepened. Mocks can be useful to fake some modules that cannot be tested. CppUTest has support for building mocks, which is suitable for embedded software. A mock could be created to fake one of Mafalda dependencies, for example, or to interact with the Led Driver module.

Still in the context of TDD, it would be interesting to explore more stages in the embedded TDD cycle seen in Figure 4.3, using Mafalda and Led Driver projects. At least to reach the stage three to run unit tests from these projects in evaluation boards and add the test runs into the continuous integration build.

References

[1] "Exatronic :: Home." [Online]. Available: http://exatronic.pt/en/home/. [Accessed: 16-May-2015].

[2] K. Pohl, *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Berlin Heidelberg, 2010, p. 814.

[3] J. A. Highsmith, *Agile Software Development Ecosystems*. Addison-Wesley Professional, 2002, p. 404.

[4] J. Verner and M. a. Babar, "Software quality and agile methods," *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf. 2004. COMPSAC 2004.*, pp. 520–525, 2004.

[5] M. Fowler and J. Highsmith, "The agile manifesto," *Softw. Dev.*, vol. 9, no. 8, pp. 28–35, 2001.

[6] C. Larman, *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2004, p. 342.

[7] E. Whitworth and R. Biddle, "Motivation and Cohesion in Agile Teams," in *Agile Processes in Software Engineering and Extreme Programming*, 2007, pp. 62–69.

[8] P. Laplante and C. Neill, "The demise of the waterfall model is imminent," *Queue*, vol. 1, no. 10, 2004.

[9] W. Royce, "Managing the development of large software systems," *Proc. IEEE WESCON*, vol. 26, no. 8, pp. 328–388, 1970.

[10] K. Petersen, C. Wohlin, and D. Baca, "The Waterfall Model in Large-Scale," in *Product-focused software process improvement*, Springer Berlin Heidelberg, 2009, pp. 386–400.

[11] R. Victor, "Iterative and incremental development: A brief history," *IEEE Comput. Soc.*, vol. 6, pp. 47–56, 2003.

[12] K. Waters, "Agile Development Cycle," 2011. [Online]. Available: http://www.allaboutagile.com/agile-development-cycle/. [Accessed: 23-Jun-2015].

[13] N. B. Moe, T. Dingsøyr, and T. Dybå, "A teamwork model for understanding an agile team: A case study of a Scrum project," *Inf. Softw. Technol.*, vol. 52, no. 5, pp. 480–491, May 2010.

[14] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review.," *Inf. Softw. Technol.*, vol. 50, no. 9–10, pp. 833–859, Aug. 2008.

[15] G. Melnik and F. Maurer, "Comparative Analysis of Job Satisfaction in Agile and Non-agile Software Development Teams," in *Extreme Programming and Agile Processes in Software Engineering*, Springer Berlin Heidelberg, 2006, pp. 32–42.

[16] A. Begel and N. Nagappan, “Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study,” *Empir. Softw. Eng. Meas.*, pp. 255–264, Sep. 2007.

[17] O. Salo and P. Abrahamsson, “Agile methods in European embedded software development organisations: a survey on the actual use and usefulness of Extreme Programming and Scrum,” *IET Softw.*, vol. 2, no. 1, pp. 58–64, 2008.

[18] D. J. Power, A. S. Sohal, and S.-U. Rahman, “Critical success factors in agile supply chain management - An empirical study,” *Int. J. Phys. Distrib. Logist. Manag.*, vol. 31, no. 4, pp. 247–265, 2001.

[19] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd Ed. Addison-Wesley Professional, 2004, p. 224.

[20] “Extreme Programming: A Gentle Introduction.” [Online]. Available: http://www.extremeprogramming.org/.

[21] “Guide to Agile Practices.” [Online]. Available: http://guide.agilealliance.org/.

[22] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Pearson Education, 2009, p. 504.

[23] S. Nerur, R. Mahapatra, and G. Mangalaraj, “Challenges of migrating to agile methodologies,” *Commun. ACM*, vol. 48, no. 5, pp. 72–78, 2005.

[24] K. Schwaber and J. Sutherland, “The scrum guide,” *Scrum Alliance*, 2011.

[25] J. Stapleton, *DSDM: Business focused development*, 2nd Ed. Pearson Education, 2003, p. 272.

[26] A. Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*. Pearson Education, 2004, p. 336.

[27] S. R. Palmer and M. Felsing, *A Practical Guide to Feature-Driven Development*. Pearson Education, 2001, p. 271.

[28] M. Poppendieck, “Lean Software Development,” in *29th International Conference on Software Engineering (ICSE’07 Companion)*, 2007, pp. 165–166.

[29] M. dos S. Soares, “Metodologias Ágeis extreme programming e scrum para o desenvolvimento de software,” *Rev. Eletrônica Sist. Informação*, vol. 3, no. 1, 2004.

[30] H. Kniberg, “Scrum and XP from the Trenches,” *How we do Scrum*, 2006.

[31] R. S. Pressman, *Software engineering: a practitioner’s approach*, Seventh Ed. the University of California: McGraw-Hill Higher Education, 2010, p. 928.

[32] M. Paasivaara, S. Durasiewicz, and C. Lassenius, “Using Scrum in Distributed Agile Development: A Multiple Case Study,” in *2009 Fourth IEEE International Conference on Global Software Engineering*, 2009, pp. 195–204.

[33] Shibu, *Introduction to Embedded Systems*. Tata McGraw Hill Education Private Limited, 2009, p. 740.

[34] L. D. Eggermont, "Embedded Systems Roadmap 2002," *STW Technol. Found. Utrecht, Netherlands*, 2002.

[35] B. Greene, "Agile Methods Applied to Embedded Firmware Development," in *Agile Development Conference*, 2004, pp. 71–77.

[36] D. D. Gajski and F. Vahid, "Specificatin and Design of Embedded Hardware-Software Systems," *IEEE Des. Test Comput.*, vol. 12, no. 1, pp. 53–67, 1995.

[37] M. Barr and A. Massa, *Programming Embedded Systems: With C and GNU Development Tools*. "O'Reilly Media, Inc.," 2006, p. 336.

[38] B. Craft, *First Steps with Embedded Systems*. 2012, p. 228.

[39] E. A. Lee, "What are the Key Challenges in Embedded Software?," *Syst. Des. Front.*, vol. 2, no. 1, 2005.

[40] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 461–491, 2004.

[41] M. Smith, J. Miller, L. Huang, and A. Tran, "A More Agile Approach to Embedded System Development," *IEEE Softw.*, vol. 26, no. 3, pp. 50–57, May 2009.

[42] W. Wolf, "Hardware-Software Co-Design of Embedded Systems," *Proc. IEEE*, vol. 82, no. 7, pp. 967–989, 1994.

[43] B. Graaf, M. Lormans, and H. Toetenel, "Embedded Software Engineering : The State of the Practice," *Software, IEEE*, vol. 20, no. 6, pp. 61–69, 2003.

[44] A. A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *Computer (Long. Beach. Calif).*, vol. 38, no. 2, pp. 63–69, Feb. 2005.

[45] Frank Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2006, p. 286.

[46] P. Ashenden, *The designer's guide to VHDL*, 3 edition. Morgan Kaufmann, 2010, p. 936.

[47] G. Booch, J. Rumbaugh, and I. Jacobson, *UML: guia do usuário*. Elsevier Brasil, 2006, p. 474.

[48] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Des. Test Comput.*, vol. 18, no. 6, pp. 23–33, 2001.

[49] L. Cordeiro, R. Barreto, R. Barcelos, M. Oliveira, V. Lucena, and P. Maciel, "Agile Development Methodology for Embedded Systems: A Platform-Based Design Approach," in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, 2007, pp. 195–202.

[50] J. Srinivasan, R. Dobrin, and K. Lundqvist, "'State of the Art' in Using Agile Methods for Embedded Systems Development," in *33rd Annual IEEE International Computer Software and Applications Conference*, 2009, pp. 522–527.

[51] P. Manhart and K. Schneider, "Breaking the ice for agile development of embedded software: an industry experience report," in *26th International Conference on Software Engineering*, 2004, pp. 378–386.

[52] B. Fitzgerald, G. Hartnett, and K. Conboy, "Customising agile methods to software practices at Intel Shannon," *Eur. J. Inf. Syst.*, vol. 15, no. 2, pp. 200–213, Apr. 2006.

[53] S. H. VanderLeest and A. Buter, "Escape the waterfall: Agile for aerospace," in *IEEE/AIAA 28th Digital Avionics Systems Conference*, 2009, p. 6–D.

[54] "Atmel® Studio 6 - Supporting Two Architectures: AVR and ARM, with One Integrated Studio - Overview." [Online]. Available: http://www.atmel.com/microsite/atmel_studio6/. [Accessed: 16-May-2015].

[55] "Cpputest." [Online]. Available: http://cpputest.github.io/. [Accessed: 31-Jul-2014].

[56] "Manifesto for Agile Software Development." [Online]. Available: http://agilemanifesto.org/.

[57] "Cygwin User's Guide." [Online]. Available: https://cygwin.com/cygwin-ug-net/cygwin-ug-net.html. [Accessed: 04-Oct-2014].

[58] "Jenkins Wiki." [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins. [Accessed: 24-Jun-2015].

[59] "PMD - Finding copied and pasted code." [Online]. Available: http://pmd.sourceforge.net/pmd-4.2.5/cpd.html. [Accessed: 20-Aug-2014].

[60] "Cppcheck 1.66 Manual." [Online]. Available: http://cppcheck.sourceforge.net/manual.pdf. [Accessed: 04-Oct-2014].

[61] "Warnings Plugin - Jenkins - Jenkins Wiki." [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Warnings+Plugin. [Accessed: 17-May-2015].

[62] "Task Scanner Plugin - Jenkins - Jenkins Wiki." [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Task+Scanner+Plugin. [Accessed: 17-May-2015].

[63] J. Grenning, *Test-Driven Development for Embedded C*. Pragmatic Bookshelf, 2011.

[64] "ArticleS.UncleBob.TheThreeRulesOfTdd." [Online]. Available: http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd. [Accessed: 19-May-2015].

[65] J. Grenning, "Agile Embedded Software Development," *Embed. Syst. Conf.*, vol. 349, pp. 1–19, 2007.

[66] P. Fonseca, “Unit Testing in Embedded Systems based on ATMEL Micro- Controllers Atmel Software Framework,” 2013.

[67] S. Chacon and B. Straub, *Pro Git*, 2nd ed. Apress, 2014, p. 456.

[68] “Git branch | Atlassian Git Tutorial.” [Online]. Available: https://www.atlassian.com/git/tutorials/using-branches/git-branch. [Accessed: 04-Jun-2015].

[69] R. Stallman, “Using and porting the GNU compiler collection,” *MIT Artif. Intell. Lab.*, 2001.

[70] “xUnit Plugin - Jenkins - Jenkins Wiki.” [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin. [Accessed: 03-Mar-2015].

[71] “DRY Plugin - Jenkins - Jenkins Wiki.” [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/DRY+Plugin. [Accessed: 24-Aug-2014].

[72] “Claim plugin - Jenkins - Jenkins Wiki.” [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Claim+plugin. [Accessed: 21-May-2015].

[73] “Email-ext plugin - Jenkins - Jenkins Wiki.” [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Email-ext+plugin. [Accessed: 21-May-2015].

[74] M. Morales, M. J. Palma, F. Pulskamp, S. Rau, M. Venkatesan, and A. Dugar, “Intelligent Systems: The Next Big Opportunity,” 2011.

Annexes

Annex A

Study results and Recommendations

The next tables have the results and recommendations of studies made in embedded projects using agile development practices.

Works	TDD	Continuous Integration
Firmware for a family of Intel processors [35].	"Become a tool to support the coding itself. It forced the team to think about what the code should be doing before coding."	"This was never a problem for us (...) forces people to integrate changes from the repository into their private workspaces before they can commit changes, so integration is done every couple of days or so with no bad side effects."
The development of the pulse oximeter equipment [49].	What they done: "(...) first write the unit test for the functionality, compile the unit test before really writing the functionality"	
Agile on Embedded system, recommendations [50].	“Increases the overall quality of the development software. Helps to elicit design decisions that were made as part of the hardware development process."	
A European wide research focusing on the research and deployment of agile methods in embedded software development [17].	One of the least used on companies adopting XP. "TDD (...) are likely to require more fundamental and extensive changes in the development approach, technologies and mindset."	According to the survey, is one of the five practices most used on companies adopting XP.
Study of software practices at Intel Shannon [52].	"Helped developers get a better understanding of what functionality was required of the software from a client point of view."	"(…) is practiced for each component, given the complexity of overall software and the need for external test equipment, full system integration is done only in the fortnight leading up to a release."
Pilot studies on agile techniques in the aerospace domain [53].	"(...) test developers are involved in the development of requirements (...) This mitigates the risk of requirements changes due to un-testable requirements being identified during the testing phase of a program."	"(...) has allowed our teams to immediately identify any issues where a change to one component impacted another component. This has been extremely beneficial in identifying areas of functionally which have been unintentionally left public and have therefore been used by other components."

Works	Frequent / Small Releases	Refactoring
Firmware for a family of Intel processors [35].		"Now that we have a name for it, this activity occurs more often and people look for opportunities to improve the quality of existing code. This awareness had reduced the amount of “kludginess” introduced in coding, and is slowly improving the quality of legacy code."
The development of the pulse oximeter equipment [49].		"(…) the application of this process would lead to elimination of duplicated code, reduction of the amount of system’s functions, and improve the system performance."
Agile on Embedded system, recommendations [50].		
A European wide research focusing on the research and deployment of agile methods in embedded software development [17].		"The two XP practices that gained ‘I do not know’ responses were refactoring and simple design. However, on the basis of the survey, it cannot be evaluated whether this was related to the unfamiliarity of the practices, lack of experience with them, or both."
Study of software practices at Intel Shannon [52].	"(...) is not feasible early in the product schedule as software releases are tied to silicon availability. Once silicon is available the team typically delivers minor releases every 4 to 6 weeks (...)"	"They found it worked best when done early, as it eliminated a lot of bugs, which would have taken up a lot of debugging time otherwise."
Pilot studies on agile techniques in the aerospace domain [53].		Not used, but it is a future work. “The containment of refactoring is important in a DO-178 environment because an apparently minor modification to one section of source code could have major impact on requirements documents, design documents, requirements based tests, or systems tests."

Works	Collective Ownership	Customer feedback / On-site customer
Firmware for a family of Intel processors [35].	"Ownership is hard to give up. People always gravitate to the area they have domain expertise in, and they feel responsible for progress in this area. However, engineers have accepted that others may improve their code and don't get upset when it happens."	"(...) the PAL firmware architecture is already defined, there is no external customer to negotiate with." "(...) the technical lead, (...) was able to make recommendations for prioritization of deliverable functionality as a customer proxy. (...) hardware design team is frequently requesting firmware changes to support the processor design, and our team is collocated with and closely integrated organizationally with the hardware design team. Sitting among the design team promoted good interaction and early discussion of alternatives."
The development of the pulse oximeter equipment [49].		
Agile on Embedded system, recommendations [50].	"Knowledge transfer and sharing are foundational to successful agile adoption. It is very difficult for agile teams to operate in a 'closed' environment that does not incentivize open knowledge sharing and collective ownership of the developed software. "	
A European wide research focusing on the research and deployment of agile methods in embedded software development [17].	One of the most used and appreciated XP practices, with no negative experiences, according to the survey.	"(...) the most non-applicable XP practice among the respondents was the on-site customer." One of the least used on companies adopting XP. "(...) are likely to require more fundamental and extensive changes in the development approach, technologies and mindset."
Study of software practices at Intel Shannon [52].	"In the past (...) developers had to choose between bringing any code they wrote with them and continuing to maintain it, or spending time teaching the code to someone else and handing over responsibility." "(...) ensure that several members of the project team knew the code well enough to make changes, and if one person was busy, another person could make the requested change." "However, (...) was only appropriate on a single-team basis. Code ownership across multiple teams was not applied."	"Unused as in early conceptual stages of development there are no specific customer." "The product marketing group act as a customer proxy, prioritizing features based on potential revenues."
Pilot studies on agile techniques in the aerospace domain [53].	This practice was not included on pilot studies.	"The clients have been involved both on a daily basis (aware of our activities and responding to questions) as well as on iteration boundaries to determine the deliverables for the next iteration. This involvement and flexibility has allowed us to change the focus of our teams easily to be able to help meet the requirements of our clients."

Works	Product Backlog	Daily meetings
Firmware for a family of Intel processors [35].		"(...) highly effective at making team members aware of what others in the group are doing, and provides great feedback for the manager also." "(...) keeps the group focused on the right things; people don't get distracted for too long on side projects or stuck on blocking issues." "However, team members grew tired of the daily meetings until we shifted the focus to "what are you working on today and what the roadblocks are?" and de-emphasized the question "what did you do yesterday?"" "(...) were taking too much time and were not adding value to their work, so we kept this part brief."
The development of the pulse oximeter equipment [49].		"(...) provide a great feedback to the product leader and create the habit of sharing the knowledge."
Agile on Embedded system, recommendations [50].		
A European wide research focusing on the research and deployment of agile methods in embedded software development [17].	"The Product Backlog seems to be a most favored Scrum practice." "(...) the concept of Product Backlog was not strictly associated with Scrum and, thus, it could have been considered as a more universal concept for managing requirements in projects."	One of the most used scrum practices.
Study of software practices at Intel Shannon [52].		
Pilot studies on agile techniques in the aerospace domain [53].		

Works	Coding Standards	Sprints
Firmware for a family of Intel processors [35].	This practice is already required by the company.	"(...) it allowed them to commit to completing well-understood tasks in a reasonable time frame while still allowing flexibility in responding to changes."
The development of the pulse oximeter equipment [49].	This practice is already required by the company.	
Agile on Embedded system, recommendations [50].		
A European wide research focusing on the research and deployment of agile methods in embedded software development [17].	Is among the most appreciated and used XP practices. “(...) it cannot be estimated whether the respondents have applied each practice as part of XP. (...) can be applied in any process model of software development whether agile or traditional."	One of the least used scrum practices.
Study of software practices at Intel Shannon [52].		"The sprint protects the team from the environment surrounding it for a meaningful amount of time."
Pilot studies on agile techniques in the aerospace domain [53]		

Works	Pair Programming
Firmware for a family of Intel processors [35].	"(...) in assembly language had some value during early stages of the design and coding, but that there came a point where it was not efficient." "(...) we now advocate pairing during detailed design and initial coding, and then splitting up once the coding gets tedious." "(...) we left it up to the developers to decide when they had reached the point where having two people code assembly instructions loses its effectiveness."
The development of the pulse oximeter equipment [49].	
Agile on Embedded system, recommendations [50].	
A European wide research focusing on the research and deployment of agile methods in embedded software development [17].	"(...) the least appreciated of the applied XP practices." One of the XP practices that gained most 'never applied'.
Study of software practices at Intel Shannon [52].	"(...) code quality level was achieved earlier." "(...) pair-programmed components had the lowest defect density in the whole product." "Developers also believe that they learned quite a lot from each other and that they remained more focused on the job at hand, less likely to go off on a tangent." "(...) also ensured that more than one developer gained a deep understanding of the design and code, thus facilitating collective ownership" "However, (...) it was found to be unsuitable for simple or well-understood problems, which could be fixed as quickly as a single developer could type." "Some developers found Pair-Programming could break their flow of concentration as they needed to pause to communicate non-obvious ideas to the pair partner." "(...) that it was difficult to reflect and concentrate with someone by their side."
Pilot studies on agile techniques in the aerospace domain [53]	"The involvement of both engineers ensured that the fix was implemented properly. An individual engineer working the same task would likely have produced an implementation with more defects, and spent more calendar time implementing the solution." "(...) one novice engineer and one experienced engineer (...) the novice engineers are able to learn the system and the process while still providing value to the program. The experienced engineers often gained (...) because they were forced to answer questions that they might not have considered before."

Annex B

Code Example to test the Static Analysis Tools

Cppcheck

The next example was made in order to throw some issues. It does not do something in specific, it is a dumb code just to use as an example to test Cppcheck, a static analysis tool.

```
#include "CheckErrors.h"
#include <stdio.h>
#include <string.h>

#define bool int
#define true 1
#define false 0

void CheckErrors_Create(void){}

void CheckErrors_Destroy(void){}

bool z(int x)
{
  int i;
  if (x == 0)
  {
    i = 0;
    return false;
  }
  return true;
}

char* createA()
  {
        char a[100];
        return a;
  }

void destroyA(void* p)
  {
        free(p);
        int z = 10;
        free(p);
  }

void s(int x)
{
    char* f = createA();
    if (x == 1)
        return;
    destroyA(f);
}
```

```
int a()
{
        int i = 0;
        char a[10];
        char b[20];

        a[10] = 0;

        for (i = 0; i<20; i++)
        {
                b[i] = a[i];
        }

        return 0;
}
```

Copy/Paste Detector (CPD)

The above two methods, each on from different files, have pieces of code with unnecessary duplication and both were analyzed using the CPD tool.

```
#include "CheckDuplCode.h"
#include "dumbClass.h"

int bbb(int b)
{

        char strOK[5]    = "OK;\n";
        char strNOK[7]    = "NOK;\n";
        char strNA[7]    = "N/A;\n";
        char strAux1[6]    = "Teste";
        char strSimul[11]  = "Simulacao ";
        char buffer[6];

        int result = 0;
        int var = fetch();

        switch(var) {
                case 1:
                        type(strAux1);
                        send(strOK);
                        result = aaa(var);
                break;

                case 2:
                        type(strAux1);
                        send(strOK);
                        result = aaa(var+1024);
                break;
        }
```

```
#include "CheckDuplCode.h"
#include "dumbClass.h"

int aaa(int b)
{
        int c = 3;
        int r = 0;
        int i = 0;
        int abc = 0;
        int x = c + r;
        int axss = 7;

        abc = b + c;

        char strOK[5]    = "OK;\n";
        char strNOK[7]    = "NOK;\n";
        char strNA[7]    = "N/A;\n";
        char strAux1[6]    = "Teste";
        char strSimul[11]  = "Simulacao ";
        char buffer[6];

        switch(abc) {
                case 1:
                        type(strAux1);
                        send(strOK);
                        x = c + r;
                        axss = 7;
                break;

                case 2:
                        type(strAux1);
                        send(strNOK);
                break;

                case 3:
                        type(strSimul);
                        send(strNA);
                        c = r + i + c;
                break;
        }

        for(i = 0; i<=10; i++)
        {
                r = (abc + i) - c;
                add(buffer);
        }

        return r;
}
```

The following files are an example of a solution to resolve the issues found by CPD. It is possible to observe some style issues that were easily refactored.

```
#ifndef D_CheckDuplCode_H
#define D_CheckDuplCode_H

struct dumbVars
        {
                char strOK[5]    = "OK;\n";
                char strNOK[7]    = "NOK;\n";
                char strNA[7]    = "N/A;\n";
                char strAux1[6]    = "Teste";
                char strSimul[11]  = "Simulacao ";
                char buffer[6];
        } vars;

#endif
```

```
#include "CheckDuplCode.h"
#include "dumbClass.h"

int aaa(int b)
{
	int c = 3;
	int r = 0;
	int i = 0;
	int abc = 0;
	int x = c + r;
	int axss = 7;

	abc = b + c;

	switch(abc) {
		case 1:
			type(vars.strAux1);
			send(vars.strOK);
			x = c + r;
			axss = 7;
		break;

		case 2:
			type(vars.strAux1);
			send(vars.strNOK);
		break;

		case 3:
			type(vars.strSimul);
			send(vars.strNA);
			c = r + i + c;
		break;
	}

	for(i = 0; i<=10; i++)
	{
		r = (abc + i) - c;
		add(vars.buffer);
	}


	return r;
}
```

```
#include "CheckDuplCode.h"
#include "dumbClass.h"

int bbb(int b)
{

	int result = 0;
	int var = fetch();

	type(vars.strAux1);
	send(vars.strOK);

	if(var == 1)
	{
		result = aaa(var);
	}
	else
	{
		result = aaa(var+1024);
	}
}
```

Annex C

CppUTest Information and Examples

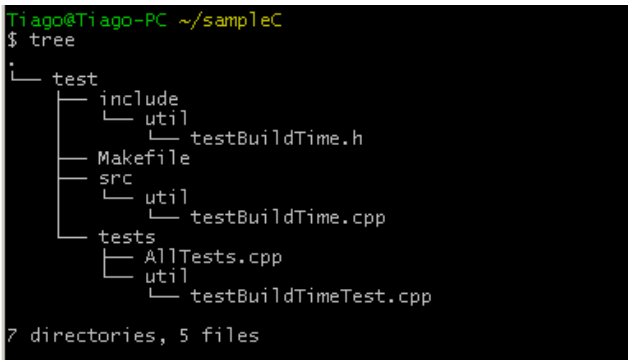
CppUTest Scripts

The TDD purpose is to start writing tests before writing code. To follow this practice, CppUTest provides a useful set of scripts that create the structure for future C projects taking the possibility to create, compile and execute tests into account. To be able to use the scripts in the shell like they were a command, first, it is necessary to run the installation script ***./InstallScripts.sh*** located inside the CppUTest scripts folder. Secondly, to execute the scripts it is necessary to define an environment variable. These variables are used to configure the user environment. They control the behavior of the shell scripts and other programs, helping them to know file locations and other things. In Linux, they are used, for example, to send information about the current working copy. Some operating systems allow the addition of new environment variables. To do this in Linux and Cygwin it is necessary to add a single line code in the file ***.bash_profile*** with the variable name and its value. In Cygwin this file is located in the user home directory. The environment variable name is ***CPPUTEST_HOME*** and the value is the path to the CppUTest folder. It was added the following line code.

```
export CPPUTEST_HOME = ~/path/to/CppUTEst/Folder/cpputest/
```

The export command makes the environment variable available in subshells, in other words, the command creates the variable to be seen by other scripts and programs invoked by the current program.

For example, using the script ***NewProject*** it will be created a structure that includes three directories, the include folder to the header files, one folder to the source code and the test folder to the test files. Inside of this folders there is a package with example files to understand how CppUTest works, giving few test examples. This package is named *util* and it is written in C++ language. There is also a makefile, created by the script, configured to compile the created project and to execute the tests. It can be modified to build the project with all the necessary conditions. The next figure presents the structure of the created project.



CppUTest Assertions

These are all the conditional checks provided by the framework CppUTest:

- CHECK(boolean condition) - checks any *boolean* result.
- CHECK_TEXT(boolean condition, text) - checks any *boolean* result and prints text on failure.
- CHECK_EQUAL(expected, actual) - checks for equality between entities using ==.
- CHECK_THROWS(expected_exception, expression) - checks if expression throws expected_exception. This check condition is only available if CppUTest is built with the standard C++ library.
- STRCMP_EQUAL(expected, actual) - check const char* strings for equality using strcmp().
- LONGS_EQUAL(expected, actual) - Compares two numbers.
- BYTES_EQUAL(expected, actual) - Compares two numbers, eight bits wide.
- POINTERS_EQUAL(expected, actual) - Compares two pointers.
- DOUBLES_EQUAL(expected, actual, tolerance) - Compares two doubles within some tolerance.
- FAIL(text) - always fails.

Additional Info about the MakefileWorker

The above piece of code is extracted from the ***MakefileWorker*** and explains how to use the available macros.

```
# MakefileWorker.mk
#
# Include this helper file in your makefile
# It makes
#    A static library
#    A test executable
#
# See this example for parameter settings
#    examples/Makefile
#
#----------
# Inputs - these variables describe what to build
#
#   INCLUDE_DIRS - Directories used to search for include files.
#                  This generates a -I for each directory
#   SRC_DIRS - Directories containing source file to built into the library
#   SRC_FILES - Specific source files to build into library. Helpful when not all
code in a directory can be built for test (hopefully a temporary situation)
#   TEST_SRC_DIRS - Directories containing unit test code build into the unit test
runner. These do not go in a library. They are explicitly included in the test
runner
#   TEST_SRC_FILES - Specific source files to build into the unit test runner
#      These do not go in a library. They are explicitly included in the test
runner
#   MOCKS_SRC_DIRS - Directories containing mock source files to build into the
test runner
#                            These do not go in a library. They are explicitly
included in the test runner
#----------
# You can adjust these variables to influence how to build the test target
# and where to put and name outputs
# See below to determine defaults
#   COMPONENT_NAME - the name of the thing being built
#   TEST_TARGET - name the test executable. By default it is
#                     $(COMPONENT_NAME)_tests
#             Helpful if you want 1 > make files in the same directory with
different
#             executables as output.
#   CPPUTEST_HOME - where CppUTest home dir found
#   TARGET_PLATFORM - Influences how the outputs are generated by modifying the
#       CPPUTEST_OBJS_DIR and CPPUTEST_LIB_DIR to use a sub-directory under the
#       normal objs and lib directories.  Also modifies where to search for the
#       CPPUTEST_LIB to link against.
#   CPPUTEST_OBJS_DIR - a directory where o and d files go
#   CPPUTEST_LIB_DIR - a directory where libs go
#   CPPUTEST_ENABLE_DEBUG - build for debug
#   CPPUTEST_USE_MEM_LEAK_DETECTION - Links with overridden new and delete
#   CPPUTEST_USE_STD_CPP_LIB - Set to N to keep the standard C++ library out
#             of the test harness
#   CPPUTEST_USE_GCOV - Turn on coverage analysis
#             Clean then build with this flag set to Y, then 'make gcov
```

```
#   CPPUTEST_MAPFILE - generate a map file
#   CPPUTEST_WARNINGFLAGS - overly picky by default
#      OTHER_MAKEFILE_TO_INCLUDE - a hook to use this makefile to make
#             other targets. Like CSlim, which is part of fitnesse
#      CPPUTEST_USE_VPATH - Use Make's VPATH functionality to support user
#             specification of source files and directories that aren't below
#             the user's Makefile in the directory tree, like:
#                    SRC_DIRS += ../../lib/foo
#             It defaults to N, and shouldn't be necessary except in the above
case.
#----------
##  Other flags users can initialize to sneak in their settings
#      CPPUTEST_CXXFLAGS - flags for the C++ compiler
#      CPPUTEST_CPPFLAGS - flags for the C++ AND C preprocessor
#      CPPUTEST_CFLAGS - flags for the C complier
#      CPPUTEST_LDFLAGS - Linker flags
```

Using an example to test CppUTest

The code below was created to be tested. It is a program to calculate the factorial of positive integer numbers.

```
#include "factorial.h"

long factorial(int n)
{
  if(n<0)
  {
    return 0;
  }

  int i;
  long result = 1;

  for (i = 1; i <= n; i++)
  {
    result = result * i;
  }

  return result;
}
```

The code in the following page contains a set of tests created to test the factorial program.

```
#include "CppUTest/TestHarness.h"
#include "factorial.h"

TEST_GROUP(factorial)
{

        void setup()
        {
        }
        void teardown()
        {
        }
};

TEST(factorial, inpuNegative)
{
        int input = -1;
        long result = factorial(input);
        LONGS_EQUAL(0, result);
}

TEST(factorial, testRandomFactorial)
{
        int input = 3;
        long result = factorial(input);
        LONGS_EQUAL(6, result);
}

TEST(factorial, factorialOfZero)
{
        int input = 0;
        long result = factorial(input);
        LONGS_EQUAL(1, result);
}
```

Now comes the makefile to compile the tests. It is small because it imports the ***MakefileWorker*** from CppUTest.

```
COMPONENT_NAME = Factorial_Test
CPPUTEST_HOME = C:\Users\Tiago\Desktop\Local_Test\cpputest
CPP_PLATFORM = gcc

SRC_DIRS = \
        src \

TEST_SRC_DIRS = \
        tests \

INCLUDE_DIRS =\
        incl \
        $(CPPUTEST_HOME)/include  \

CPPUTEST_WARNINGFLAGS =
CPPUTEST_WARNINGFLAGS +=
CPPUTEST_CFLAGS =
CPPUTEST_CXXFLAGS +=

include $(CPPUTEST_HOME)/build/MakefileWorker.mk
```

Annex D

Led Driver Project Makefiles

I created two makefiles to compile James Greening Led Driver project using CppUTest and the Atmel studio headers. The first one was created from the scratch, the second one imports the MakefileWorker. They both run only in Windows operating systems.

```
#################################################################
########################INCLUDE_DIRS#############################

INCLUDE_DIRS =\
Led_Tests \
Led_Driver/incl \
Led_Driver/mocks \
Led_Tests/avr_tests \
Led_Tests/atmel_Lib_Headers

#################################################################

INCLUDES = $(foreach d, $(INCLUDE_DIRS), -I$d)

#################################################################
#######################SOURCE_TEST_DIRS##########################
SRC_DIRS = \
Led_Tests \
Led_Driver/src \

#################################################################

get_src_from_dir  = $(wildcard $1/*.c) $(wildcard $1/*.cpp)
get_src_from_dir_list = $(foreach dir, $1, $(call get_src_from_dir,$(dir)))
SOURCES += $(call get_src_from_dir_list, $(SRC_DIRS))

OBJ_DIR = objs

__src_to = $(subst .c,$1, $(subst .cpp,$1,$2))
src_to = $(addprefix $(OBJ_DIR)/,$(call __src_to,$1,$2))
src_to_o = $(call src_to,.o,$1)
OBJECTS = $(call src_to_o,$(SOURCES))
#################################################################
#############################INPUTS##############################
COMPONENT_NAME = Led_Test
CPPUTEST_HOME = C:\Users\Tiago\Desktop\Local_Test\cpputest

AVRDEVICE=AVR_ATmega32

CC = gcc

#CFLAGS for C
CFLAGS = -std=c99
```

```
#CXXFLAGS for c++
CXXFLAGS =

#CPPFLAGS FOR c and c++
CPPFLAGS = -D__$(AVRDEVICE)__
CPPFLAGS += -DCPPUTEST
CPPFLAGS += -I$(CPPUTEST_HOME)/include

LDLIBS = -lCppUTest -lCppUTestExt -lpthread -lstdc++
LDFLAGS = -L$(CPPUTEST_HOME)/lib

################################################################

all:  $(COMPONENT_NAME) run

$(COMPONENT_NAME): make_obj_dir $(OBJECTS)
        @echo
        @echo $(OBJECTS)
        @echo Linking $(COMPONENT_NAME)
        g++ $(OBJECTS) $(CPPFLAGS) $(LDFLAGS) -o $@ $(LDLIBS)

$(OBJ_DIR)/%.o: %.cpp
        @echo Compiling $(notdir $<)
        g++ $(CPPFLAGS) $(CXXFLAGS) $(INCLUDES) -c $< -o $@

$(OBJ_DIR)/%.o: %.c
        @echo Compiling $(notdir $<)
        gcc $(CPPFLAGS) $(CFLAGS) $(INCLUDES) -c $< -o $@

make_obj_dir:
        @$(call make_obj_dir)

run: $(COMPONENT_NAME)
        ./$(COMPONENT_NAME)

clean:
        $(RM) -R $(COMPONENT_NAME) $(OBJ_DIR)

# Create obj directory structure
define make_obj_dir
        mkdir -p $(OBJ_DIR);
        for dir in $(SRC_DIRS); \
        do \
```

The previous makefile is more complex, the developer has more control about the compilation. The next makefile, in the following page, imports the ***MakefileWorker*** from CppUTest source folder. It is simpler but the framework controls all the compilation process.

```
#--- Inputs ----#
COMPONENT_NAME = Led_Test
CPPUTEST_HOME = C:\Users\Tiago\Desktop\Local_Test\cpputest

CPPUTEST_USE_EXTENSIONS = Y
CPP_PLATFORM = gcc

SRC_DIRS = \
        Led_Driver \
        Led_Driver/src

TEST_SRC_DIRS = \
        Led_Tests \

INCLUDE_DIRS =\
        Led_Tests \
        $(CPPUTEST_HOME)/include  \
    Led_Driver/incl \
    Led_Driver/mocks \
    Led_Tests/avr_tests \

CPPUTEST_WARNINGFLAGS = -Wall
CPPUTEST_WARNINGFLAGS += -Wswitch-default
CPPUTEST_WARNINGFLAGS += -Wextra

CPPUTEST_CFLAGS =
CPPUTEST_CFLAGS += -D__AVR_ATmega32__
CPPUTEST_CFLAGS += -DCPPUTEST
CPPUTEST_CFLAGS +=
CPPUTEST_CFLAGS +=

CPPUTEST_CXXFLAGS += -D__AVR_ATmega32__
CPPUTEST_CXXFLAGS += -DCPPUTEST

#LD_LIBRARIES = -lpthread -lstdc++

include $(CPPUTEST_HOME)/build/MakefileWorker.mk
```